SEKI-PROJEKT

SEKI
MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany

Komponenten des Interaktiven Systems SPESY zur Unterstützung Integrierter Programm-Spezifikation und -Verifikation (Band I)

Walter Sommer

Memo SEKI-84-02-I

SPESY

EIN INTERAKTIVES SYSTEM ZUR UNTERSTÜTZUNG

INTEGRIERTER PROGRAMM-SPEZIFIKATION UND -VERIFIKATION

(Band I)

Walter Sommer

Abstract

An interactive system for the development of hierarchies of parameterized algorithmic specifications is presented. Implemented in SIEMENS-Interlisp, the system SPESY supports entering, editing, instantiating, and managing specifications written in a predecessor of the software specification language ASPIK.
Volume I presents the language by detailed examples with special emphasis to the instantiation on parameterized specifications. Moreover, Volume I contains the SPESY-manual. The system documentation of SPESY is given in Volume II. Volume III contains the source code and the original protocol of a SPESY-session with all examples presented in Volume I.

Zusammenfassung

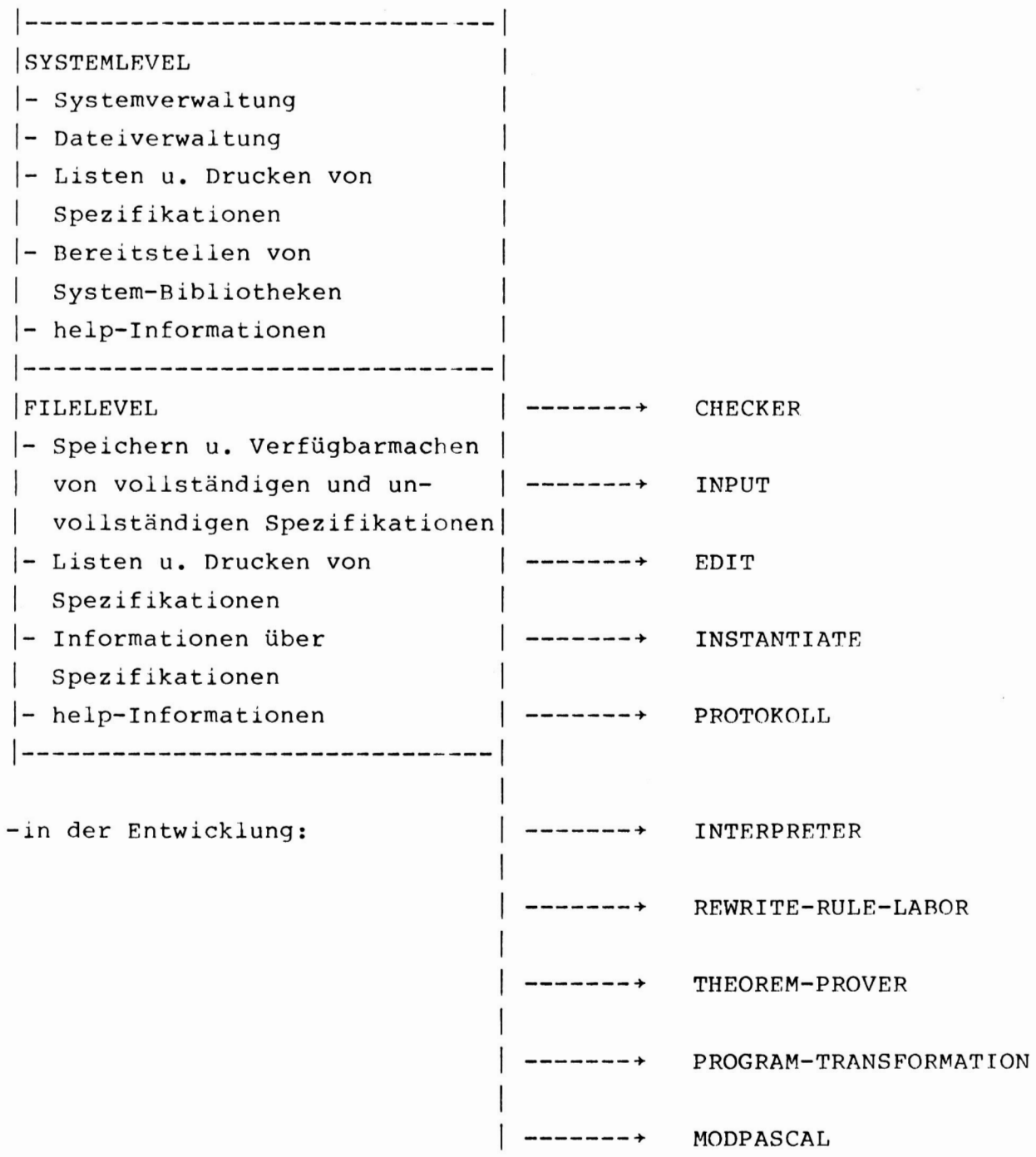
Ein interaktives System zur Entwicklung von parametrisierten, algorithmischen Spezifikations-Hierarchien wird vorgestellt. Das in SIEMENS-Interlisp implementierte System SPESY erlaubt systemunterstütztes Eingeben, Editieren, Instanziieren und Verwalten von Spezifikationen in einer Vorläufer-Version der Softwarespezifikationssprache ASPIK.
In Band I wird die Sprache anhand von ausführlichen Beispielen vorgestellt, wobei besonders auf die Instanziierung parametrisierter Spezifikationen eingegangen wird. Außerdem enthält Band I das SPESY-Benutzerhandbuch. Die Systemdokumentation ist in Band II zu finden. Band III enthält den Programmtext, sowie das Original-Protokoll einer SPESY-Sitzung mit allen in Band I aufgeführten Beispielen.

Band I

1. Einleitung

Angesichts der Komplexität und hohen Anforderungen an die Sicherheit vieler Software-Systeme erweist sich die Programmverifikation bei der Erstellung dieser Systeme als unabdingbar.
Erfolgen Problemspezifizierung und Programmierung voneinander getrennt, so liegt die Schwierigkeit beim Nachweis der Konsistenz von Spezifikation und Programm darin begründet, daß für die Verifikation oft die dem Programm-Design zugrunde liegenden Ideen nachvollzogen werden müssen. Das ist aber für mechanische Verifikationssysteme i.a. nicht möglich.
In dem an der Universität Kaiserslautern durchgeführten Forschungsprojekt ´PROGRAMMVERIFIKATION´, in dessen Rahmen auch diese Arbeit erstellt wurde, wird deshalb eine andere Vorgehensweise bei der Entwicklung von Software verfolgt. Gemäß dem VERIFY WHILE DEVELOP PARADIGM soll Software strukturiert in mehreren Ebenen entwickelt werden. Dabei erfolgt jeder Entwicklungsschritt entweder automatisch durch mechanische Werkzeuge, die die Korrektheit des Schrittes garantieren, oder aber er wird durch automatische Theorem-Beweiser verifiziert.
Das Projekt ist ausführlich im Projektantrag [RS 80], im überarbeiteten Projektantrag [RS 82] und in der Veröffentlichung INTEGRATED PROGRAM DEVELOPMENT AND VERIFICATION [SEKI 83] beschrieben.
Das Ziel des Projektes ist die Vorlage eines integrierten Systems, das einem Softwareentwickler alle für die verfolgte Verfahrensweise notwendigen Werkzeuge an einem Arbeitsplatz zur Verfügung stellt. Ausgehend von der Problemspezifizierung in der Spezifikationssprache ASPIK steht am Ende der Entwicklungsschritte ein bezüglich der Problemspezifikation verifiziertes PASCAL-Programm.
Entwickelt wurde ASPIK im Rahmen des SEKI-Projektes (siehe [BGORV 82]). Eine Vielzahl von Beispielspezifikationen sind zu finden in [GR 82] und [ST 83], in denen eine Spezifikation des industriell gefertigten Softwareprodukts INTAKT [SIEM 80] vorgelegt wird.

Kernstück des Gesamtsystems ist das in SIEMENS-INTERLISP implementierte, interaktive und modular aufgebaute Spezifikationssystem SPESY. Der compilierte Code des Systems umfaßt ca. 1.5 Mega-Byte. Das System hat folgenden Aufbau:



Davon sind folgende Teile zur Zeit implementiert:

1. System- und Dateiverwaltung.
2. List- und Druckmöglichkeiten erlauben es, ASPIK-Spezifikationen ganz oder teilweise am Bildschirm listen oder ausdrucken zu lassen. Darüber hinaus besteht die Möglichkeit, sich eine Vielzahl von Informationen über ASPIK-Spezifikationen am Bildschirm listen zu lassen.
3. Bereitstellung von System-Bibliotheken mit vordefinierten ASPIK-Spezifikationen.
4. Weitreichende help-Informationen stehen dem Benutzer an jeder beliebigen Stelle zur Verfügung.
5. Der CHECK-Modul untersucht ASPIK-Spezifikationen daraufhin, ob sie syntaktisch korrekt sind.
6. Sowohl korrekte wie auch unvollständige ASPIK-Spezifikationen können abgespeichert und wieder zugänglich gemacht werden.
7. Der INPUT-Modul erlaubt dem Benutzer die Eingabe von ASPIK-Spezifikationen in zweifacher Weise:
   - Vollunterstützte Eingabe
     NachAbschluß vordefinierter Teile wird die Eingabe auf Korrektheit hin untersucht. Werden Fehler erkannt, erfolgt eine detaillierte Fehlermeldung. Alle Eingaben werden vom System angefordert.
   - Teilunterstützte Eingabe
     Alle Eingaben werden vom System angefordert. Die Korrektheitsuntersuchung entfällt.
8. Der EDIT-Modul erlaubt bereits existierende ASPIK-Spezifikationen zu editieren.
9. Der INSTANTIATE-Modul erlaubt das vollständige oder teilweise Instanziieren parametrisierter ASPIK-Spezifikationen. Der Modul bietet dem Benutzer die Möglichkeit, bei voller System-Unterstützung einen Morphismus zwischen den Signaturen von ASPIK-Spezifikationen zu konstruieren. Die dadurch bestimmten Spezifikationen werden automatisch generiert und abgespeichert.

10. Der PROTOCOL-Modul gestattet, eine SPESY-Sitzung ganz oder teilweise protokollieren und das Protokoll automatisch ausdrucken zu lassen.
11. Anschlußmöglichkeit für weitere Werkzeuge, wie:
    - Symbolischer Interpretierer der algorithmischen ASPIK-Spezifikationen [KÜ 83]
    - Rewrite-Rule-Labor [THO 83]
    - Automatischer Theorem-Beweiser [BES 81]
    - Wissensbasiertes Programm-Transformations-System [GE 83]
    - System für die Implementierung abstrakter Datentypen durch PASCAL-Programme [OLT 83]

Die vorliegende Arbeit umfaßt davon die folgenden Punkte:

1. System- und Dateiverwaltung
2. Erstellen und Listen von Informationen über ASPIK-Spezifikationen
3. Bereitstellen der System-Bibliotheken
4. help-Informationen
5. Teile des CHECK-Moduls
6. Verwaltung der ASPIK-Spezifikationen
9. INSTANTIATE-Modul
10. PROTOCOL-Modul
11. Bereitstellen der Anschlußmöglichkeiten für Systemerweiterungen

Das gesamte System SPESY ist eine Gemeinschaftsarbeit, die im Rahmen des oben erwähnten Forschungsprojektes "PROGRAM-VERIFICATION" entwickelt und von Renate Kücke, Erich Rome, Christoph Thomas und mir implementiert wurde. Die vollständige Beschreibung des Systems ist im Benutzerhandbuch [KRST 83-1] und in der Systemdokumentation [KRST 83-2] zu finden. Die Abschnitte 3.3., 4.1.4. - 4.1.6. und 4.2.2. - 4.2.4. sind überarbeitete und vervollständigte Versionen der entsprechenden Abschnitte in [KRST 83-1] und [KRST 83-2].

Die Arbeit gliedert sich in vier Hauptteile:

1. Im Kapitel THEORETHISCHE GRUNDLAGEN werden zunächst die wichtigsten Definitionen und Sätze zur Spezifikation von abstrakten Datentypen durch initiale Algebren und kanonische Termalgebren aufgeführt. Anschließend werden die Grundlagen der Parametrisierung und insbesondere der Parametrisierung von Signatur-Hierarchien behandelt. Im Gegensatz zur Spezifikation enthält eine Signatur keine Gleichungen. Bei der Instanziierung in ASPIK wird genau der Signatur-Anteil an einer ASPIK-Spezifikation aktualisiert. Um den Nachweis zu erbringen, daß ein Signatur-Morphismus auch ein Spezifikations-Morphismus ist, sind i.a. Beweise nötig, die die Konsistenz der properties der Spezifikation zu den properties der Instanz zeigen.
2. Das Kapitel SPEZIFIKATIONEN IN ASPIK befaßt sich mit dem Aufbau und der Parametrisierung von ASPIK-Spezifikationen. Anhand eines ausführlichen Beispiels wird aufgezeigt, welche Möglichkeiten die Sprache in Verbindung mit dem Parametrisierungskonzept zur Spezifizierung abstrakter Datentypen bietet.
3. Das BENUTZERHANDBUCH beschreibt SPESY aus der Sicht eines Systembenutzers. Einer allgemeinen Systembeschreibung folgt die detaillierte Beschreibung aller dem Benutzer zur Verfügung stehenden Kommandos.
   Hieran schließt sich die Aufführung der für den Betrieb von SPESY geltenden Namenskonventionen an.
4. Die SYSTEMDOKUMENTATION umfaßt ihrerseits zwei Hauptteile:
   1. Beschreibung der
      - bei der Implementierung verwandten Modultechnik,
      - den Modulen entsprechenden Dateien,
      - intern für die Darstellung einer ASPIK-Spezifikation verwandten Datenstrukturen,
      - bei der Implementierung verwandten Konzepte.
   2. Detaillierte Beschreibung jeder einzelnen Funktion.

Im Anschluß daran wird ein Überblick gegeben über die geplanten Systemerweiterungen und Fortentwicklungen des Systems.

Der Anhang enthält den Quelltext aller Funktionen von SPESY, die Gegenstand dieser Arbeit sind.

Anmerkung: Die dieser Arbeit zugrunde liegende Version der Sprache ASPIK ist eine vorläufige. Es findet zur Zeit ein Redesign von ASPIK statt. Die entgültige Version wird kleine syntaktische Änderungen, sowie ein neues Parametrisierungskonzept aufweisen.

2.1. Spezifikation von ADT

2.1.1. Spezifikation von ADT durch initiale Algebren

2.1.1.1. Spezifikation von Algebren durch Signaturen

Formale Spezifikationsmethoden haben sich als ein wichtiges Werkzeug für die Spezifikation von Problemen erwiesen, da sie folgende Vorteile gegenüber nicht-formalen Methoden aufweisen:

- Mathematische Behandlung:
  - Korrektheit
  - Äquivalenz
  - Vollständigkeit
  - Konsistenz
  - Maschinelle Implementierung
- Eindeutige Semantik

Von den verschiedenen Methoden, Probleme formal zu spezifizieren, sind in besonderem Grade die algebraischen Methoden für die Spezifikation von Datentypen geeignet. Die Spezifikationen besitzen bei allen algebraischen Ansätzen Algebren als Modelle. Die Semantik der Datentypen sind dann alle (loser Ansatz) oder eine Teilmenge (z.B. initialer Ansatz) dieser Modelle. Algebren sind durch einige gemeinsame Merkmale gekennzeichnet. Zum einen besitzen sie Daten, die auf eine oder mehrere Datenmengen verteilt sind; zum anderen besitzen sie Operationen um auf die Daten zugreifen, bzw. um die Daten verändern zu können. Damit entsprechen sie unmittelbar den Datentypen aus den Programmiersprachen.

In diesem Kapitel sollen nun einige relevante Begriffe und Ergebnisse für die algebraische Spezifikation von ADT zusammengestellt werden. Auf notwendige Beweise wird an dieser Stelle nicht eingegangen; diese sind zu finden in [RAUL 79], [ADJ 76], [KREO 78], [KLAE 82].

Zunächst muß die Möglichkeit gegeben werden, Sorten- und Operationssymbole deklarieren zu können. Dies führt zu folgender Definition:

Definition
Eine Signatur (S,Σ) besteht aus einer Menge S von Sorten und einer Familie $\Sigma=\{\Sigma_{w,s} \mid w\in S^*, s\in S\}$ von Mengen. •

Die Signatur ist so etwas wie eine Schnittstellenbeschreibung für einen ADT, denn sie legt Namen für Datenmengen, Operationen sowie Argument- und Wertebereich von Operationen fest. Die semantische Festlegung eines Datentyps erfolgt durch die Zuordnung von Algebren.

Definition
Für eine Signatur (S,Σ) besteht eine (S,Σ)-Algebra A aus:
1. Λ $s\in S$ einer Menge A_s, die Trägermenge von S heißt.
2. Λ $s\in S$, Λ $f\in\Sigma_{\varepsilon,s}$ ausgezeichneten Elementen $f_A\in A_s$.
3. Λ $w\in S^*$, $s\in S$. Λ $f\in\Sigma_{w,s}$ Operationen $f_A:A_{s1}x...xA_{sn}\rightarrow A_s$ $w=s1...sn, w\neq\varepsilon$ •

Die Algebren müssen der durch die Signaturen bestimmten Form genügen, indem sie zu den vorgegebenen Daten- und Operationssymbolen die entsprechenden Datenmengen bzw. Operationen liefern. Durch wiederholten Aufruf der Operationssymbole lassen sich Daten von A erzeugen.

Definition
A und B seien (S,Σ)-Algebren. Dann heißt eine Familie $H:=\{h_s:A_s\rightarrow B_s\}$ von Abbildungen ein (S,Σ)-Homomorphismus, wenn folgendes gilt:
1. Λ $s\in S$. Λ $f\in\Sigma_{\varepsilon,s}$. $h_s(f_A)=f_B$.
2. Λ $w=s1...sn\in S^*, s\in S$. Λ $ai\in A_{si}$ $(1\leq i\leq n)$. $h_s(f_A(a1,...,an))=f_B(h_{s1}(a1),...,h_{sn}(an))$. •

Definition
Eine (S,Σ)-Algebra A heißt initial, wenn für jede (S,Σ)-Algebra B genau ein (S,Σ)-Homomorphismus H: A → B existiert. •

Definition
Ein (S,Σ)-Homomorphismus H: A → B heißt (S,Σ)-Isomorphismus, wenn $\Lambda\, s\varepsilon S\ h_s$ eine Bijektion ist. Gibt es einen (S,Σ)-Isomorphismus zwischen A und B, dann heißen A und B isomorph. •

Folgende elementaren Behauptungen lassen sich zeigen:
1. Initiale (S,Σ)-Algebren sind isomorph. Sie gehen also durch Umbenennung auseinander hervor.
2. Es gibt einen (S,Σ)-Homomorphismus von einer initialen Algebra in jede andere (S,Σ)-Algebra.

Faßt man nun einen ADT als die Isomorphieklasse einer initialen (S,Σ)-Algebra auf, so stellt sich die Frage, ob eine solche initiale Algebra immer existiert, und wie sie aussieht.

Definition
Die Menge der (S,Σ)-Terme einer Signatur (S,Σ) ist definiert durch:
1. $\Lambda\, s\varepsilon S.\ \Lambda\, \Sigma_{\varepsilon,s}\varepsilon\Sigma.\ \Sigma_{\varepsilon,s} \subseteq T_{\Sigma,s}$.
2. $\Lambda\, sl\ldots sn\varepsilon S^*, s\varepsilon S.\ \Lambda\, f\varepsilon\Sigma_{sl\ldots sn,s}.\ \Lambda\, ti\varepsilon T_{\Sigma,si}\ (1<i<n).$
   $f(tl,\ldots,tn) \subseteq T_{\Sigma,s}$.
   $T_{\Sigma,s}$ bildet die Menge aller Terme der Sorte s. •

Die so definierten Terme lassen sich als zusammengesetzte Aufrufe der Operationssymbole verstehen, die somit als Operationsaufrufe in Σ-Algebren auf Daten zugreifen. Zu beachten ist, daß die Σ-Terme unabhängig von Σ-Algebren rein syntaktisch definiert sind. Durch den Übergang von den formalen Operationssymbolen zu den zugeordneten tatsächlichen Operationen in den Algebren definiert jeder Term einen Wert in der Algebra.

Definition

Die Termalgebra T_Σ einer Signatur (S,Σ) ist definiert durch:

1. $\Lambda\ s\varepsilon S.\ T_{\Sigma,s}$ ist Trägermenge der Sorte s.
2. $\Lambda\ s\varepsilon S.\ \Lambda\ f\varepsilon\Sigma_{\varepsilon,s}.\ fT_\Sigma := f.$
3. $\Lambda\ w=s1\ldots sn\varepsilon S^*, s\varepsilon S.\ \Lambda\ f\varepsilon\Sigma_{w,s}.\ fT_\Sigma : T_{\Sigma,s1}x\ldots xT_{\Sigma,sn} \rightarrow T_{\Sigma,s}$ ist die durch $\Lambda\ ti\varepsilon T_{\Sigma,si}.\ fT_\Sigma(t1,\ldots,tn) := f(t1,\ldots,tn)$ definierte Funktion. •

Theorem

Für jede Signatur (S,Σ) ist die Termalgebra T_Σ eine initiale Algebra. •

Damit wäre die Frage nach Existenz und nach Aussehen einer zu einer (S,Σ)-Signatur initialen Algebra beantwortet. Die Termalgebra erfüllt aber noch weitere für einen ADT charakteristische Eigenschaften und läßt damit den initialen Ansatz innerhalb der algebraischen Methoden zur Spezifikation von ADT als besonders geeignet erscheinen:

- Alle Daten der Termalgebra (nämlich die Terme) sind operationenerzeugt, denn jeder Term ist sein eigener Operationsaufruf, da die formalen mit den konkreten Operationen übereinstimmen.
- Die Eigenschaft initial zu sein bedeutet, daß zwei initiale (S,Σ)-Algebren durch Umbenennung auseinander hervorgehen, d.h. sie sind bis auf Isomorphie eindeutig bestimmt, und bei einer Umbenennung bleibt die Semantik der Algebra erhalten. Daraus folgt, daß initiale Algebren repräsentationsunabhängig sind.

2.1.1.2. Spezifikation von Algebren mit Gleichungen

Die im vorigen Abschnitt eingeführten syntaktischen und semantischen Konzepte erlauben es, einen ADT, der durch die Isomorphieklasse der initialen (S,Σ)-Algebra definiert ist, mit einer (S,Σ)-Signatur zu spezifizieren; dennoch ist es bisher nicht möglich einen ADT mit weiteren Operationen auszustatten oder über einen bereits definierten ADT, neue zu definieren. Der Grund dafür liegt darin, daß es bisher nicht möglich ist, Eigenschaften der Operationen festzulegen, auszudrücken oder zu fordern. Gesucht werden also initiale Algebren, die bestimmten Axiomen genügen. Denkbare Darstellungen für diese Axiome wären prädikatenlogiche Ausdrücke, Ungleichungen oder Gleichungen. In diesem Ansatz wird auf Gleichungen zurückgegriffen, deren Grundelemente Terme mit Variablen sind.

Definition

(S,Σ) sei eine Signatur.

1. $\Lambda\ s\varepsilon S.$ Sei $X=\{X_s \mid s\varepsilon S\}$ eine S-sortierte, unendliche, abzählbare Menge von Variablensymbolen der Sorte s, so daß:
   1) $\Lambda\ s'\varepsilon S.\ s\neq s' \Rightarrow X_s \cap X_{s'}=\emptyset.$
   2) $X_s \cap \Sigma=\emptyset.$
2. $(S,\Sigma(X))$ bildet eine Signatur mit Variablen mit:
   $\Sigma(X):=\{\Sigma(X)_{w,s} \mid w\varepsilon S^*, s\varepsilon S\}$ mit:
   1) $\Lambda\ s\varepsilon S.\ \Sigma(X)_{\varepsilon,s}:=\Sigma_{\varepsilon,s}\cup X_s.$
   2) $\Lambda\ w\varepsilon S^*, s\varepsilon S.\ \Sigma(X)_{w,s}:=\Sigma_{w,s}.$
3. Die $\Sigma(X)$-Termalgebra $T_{\Sigma(X)}$ bezeichnet man als Σ-Algebra $T_\Sigma(X)$ oder als freie Algebra über X:
   1) $\Lambda\ s\varepsilon S.\ T_\Sigma(X)_s:=T_{\Sigma(X),s}.$
   2) $\Lambda\ w\varepsilon S^*, s\varepsilon S.\ \Lambda\ f\varepsilon\Sigma_{w,s}.\ fT_\Sigma(X):=fT_{\Sigma(X)}.$
4. Für eine Σ-Algebra A heißt die Familie von Abbildungen $B:X \to A$, $B=\{b_s \mid \Lambda\ s\varepsilon S.\ b_s:X_s\to A_s\}$ Belegung der Variablen aus X in A. •

Analog zu Σ-Termen lassen sich auch Σ-Terme mit Variablen in Σ-Algebren interpretieren, wenn man nicht nur die formalen Operati-

onssymbole durch die aktuellen Operationen ersetzt, sondern auch die Variablen mit aktuellen Werten belegt. Durch die Belegung wird ein eindeutiger Σ-Homomorphismus H^* definiert.

Theorem

Sei $(S,\Sigma(X))$ eine Signatur mit Variablen.
Zu jeder Σ-Algebra A und jeder Belegung $B:X \to A$ existiert genau ein (S,Σ)-Homomorphismus $H_B^*: T_\Sigma(x) \to A$ mit $h_s^*(x)=b_s(x)$ für alle $s \in S$, $x \in X_s$. •

Wie initiale sind auch freie Σ-Algebren bis auf Isomorphie eindeutig bestimmt. Die Terme mit Variablen geben nun ein syntaktisches Konzept, um Gleichungen und Spezifikationen mit Gleichungen zu definieren.

Definition

Sei $(S,\Sigma(X))$ eine Signatur mit Variablen.

1. Für jede Sorte $s \in S$ heißt ein geordnetes Paar $e=(L,R)$ von Σ-Termen L und R der Sorte s mit Variablen in X Gleichung der Sorte s.
2. Sei $E=\{E_s \mid s \in S\}$ eine Mengenfamilie von Gleichungen der Sorte s. Dann heißt $\langle S,\Sigma,E\rangle$ eine Spezifikation. •

Nach der Ergänzung der Signaturen um Gleichungen ist jetzt erforderlich auf der semantischen Ebene zu präzisieren, wann eine Algebra die durch die Gleichungen festgelegten Eigenschaften besitzt, bzw. wann sie die Gleichungen erfüllt.

Definition

Sei $SPEC=\langle S,\Sigma,E\rangle$ eine Spezifikation und $e=(L,R) \in E$ eine Gleichung der Sorte s, H_B^* der eindeutige Homomorphismus, der B fortsetzt. Eine (S,Σ)-Algebra A erfüllt die Gleichung $e=(L,R)$, wenn für alle Belegungen $B:X \to A$ gilt: $H_B^*(L)=H_B^*(R)$.
A ist eine (S,Σ,E)-Algebra, wenn sie alle Gleichungen aus E erfüllt. •

Die Gleichungen einer Spezifikation SPEC=⟨S,Σ,E⟩ sind also Anforderungen an Σ-Algebren. Eine Σ-Algebra erfüllt genau dann die Gleichungen, wenn die Terme $h_B{}^*{}_s(L)$ und $h_B{}^*{}_s(R)$ auf dieselben Daten der Σ-Algebra bezeichnen. Die Termalgebren gehören im allgemeinen nicht zu den Algebren, die die Gleichungen aus E erfüllen, denn Terme sind nur dann gleich, wenn sie formal übereinstimmen; dennoch gilt es, die besonderen Eigenschaften der Termalgebra wie Operationenerzeugtheit und Repräsentationsunabhängigkeit auch von einer Algebra zu fordern, welche die Gleichungen erfüllt. Dieses führt zu einem Identifizierungsprozeß, in dem die Termalgebra mit Gleichungen versehen wird. Die Gleichungen aus E induzieren für jede Sorte s eine Äquivalenzrelation auf den Termen der Sorte s. Alle zu eimen Term äquivalenten Terme werden in einer Äquivalenzklasse zusammengefaßt und somit identifiziert. Die Datenmenge der Algebra besteht dann aus der Menge aller Äquivalenzklassen aller Sorten. Die Operationen der Algebren werden über diesen Äquivalenzklassen definiert, wobei Verträglichkeit mit der Äquivalenzrelation gefordert wird.

Definition

Sei ⟨S,Σ,E⟩ eine Spezifikation.

1. Die Gleichungen E induzieren auf den Termen aus T_Σ eine Familie von Relationen $\equiv_E=\{\equiv_s \mid s\in S\}$ die definiert ist durch:
   1) $H_B{}^*(L)\equiv_s H_B{}^*(R)$ für alle $(L,R)\in E$ und $B:X \to T_\Sigma$.
   2) $\wedge\, s\in S.\ \equiv_s$ ist Äquivalenzrelation.
   3) $\wedge\, w=s1\ldots sn\in S^*,\ s\in S.\ \wedge\, f\in\Sigma_{w,s}.\ \wedge\, t_i,t_i'\in T_{\Sigma,si}\ (1\le i\le n).$
      $t_i\equiv_{si} t_i' \Rightarrow f(t_1,\ldots,t_n)\equiv_s f(t_1',\ldots,t_n')$.
   4) $\equiv_s$ sei die durch E erzeugte, kleinste Kongruenzrelation mit den Eigenschaften aus 1) bis 3).

   Bemerkung: Die so definierte Relationenfamilie ist wegen 3) eine Kongruenz.
2. Für jeden Term $T_{\Sigma,s}$ werden in der Äquivalenzklasse $[t]$ alle zu t äquivalenten Terme zusammengefaßt: $[t]:=\{t' \mid t\equiv_s t'\}$.

3. Die Menge aller Äquivalenzklassen $(T_\Sigma/\equiv_E)_s=\{[t] \mid t\varepsilon T_{\Sigma,s}\}$ für alle $s\varepsilon S$ bilden die Datenmenge der Quotiententermalgebra $T_{\Sigma,E}$, deren Operationen definiert sind durch:
$\Lambda\ w=s1\ldots sn\varepsilon S, s\varepsilon S.\ \Lambda\ f\varepsilon\Sigma_{w,s}.\ \Lambda\ [t_i]\varepsilon(T_\Sigma/\equiv_E)_{si}\ (1\leq i\leq n).$
$f_\Sigma/\equiv_E([t_1],\ldots,[t_n]):=[f(t_1,\ldots,t_n)]$. •

Theorem

$T_{\Sigma,E}$ ist eine initiale (Σ,E)-Algebra.

Faßt man nun einen ADT auf als die Isomorphieklasse der initialen (S,Σ)-Algebren, die E erfüllen (vgl. [ADJ 76]), so läßt sich zusammenfassend sagen, daß jede Spezifikation $SPEC=\langle S,\Sigma,E\rangle$ einen ADT spezifiziert, der durch die Isomorphieklasse der initialen (S,Σ,E)-Algebren definiert ist.

2.1.2. Der CTA-Ansatz zur Spezifikation von ADT

Die im letzten Abschnitt vorgestellte algebraische Spezifikationsmethode von ADT ist von einem sehr hohen Abstraktionsniveau gekennzeichnet. Oft ist es jedoch einfacher und auch natürlicher in konkreten Modellen zu denken als in Axiomensystemen. Ein anderes Konzept, basierend auf der algebraischen Methode, wurde von Klären [KL 80] und Loeckx [LO 81] vorgelegt. In der algorithmischen Spezifikationsmethode wird eine konkrete Algebra explizit definiert.

Definition
A sei eine Σ-Algebra. T_Σ sei die freie Termalgebra über Σ. Dann heißen die Trägermengen von T_Σ Herbrand-Universum über Σ. •

Bemerkung: Um übermäßiges Indizieren zu vermeiden und die Unterscheidung zu ermöglichen zwischen Elementen aus dem Herbrand-Universum und Ausdrücken, die zu solchen Elementen evaluieren, sei folgende Notation vereinbart: Ist mit einem Term ein Element der Trägermenge gemeint, trägt dieser einen * als Präfix.
$*op(t_1,...,t_n)$ kennzeichnet ein Element aus dem Herbrand-Universum, $op(t_1,...,t_n)$ bezeichnet ein Element, das man erhält, wenn die Operation op auf die Argumente $t_1,...,t_n$ angewandt wird.

Definition
Eine Σ-Algebra ist eine kanonische Termalgebra CTA wenn:
1. $\Lambda\ s\varepsilon S\ A_s \subseteq H(\Sigma,s)$ (Term-Eigenschaft)
2. $*op(t_1,...,t_n)\varepsilon A \Rightarrow t_i\varepsilon A\ (i=1...n)$ (Subterm-Eigenschaft)
3. $*op(t_1,...,t_n)\varepsilon A \Rightarrow op(t_1,...,t_n)=*op(t_1,...,t_n)$ (Konstruktor-Eigenschaft)

Damit die Möglichkeit gegeben ist, Fehler in Datentypen ausdrükken zu können, werden die Träger jeder Sorte $s\varepsilon S$ um ein Fehlerelement error.s erweitert, und entsprechende Fehlerkonstanten werden in die Signaturen aufgenommen. Für die Operationen wird gefordert, daß sie strikt bezüglich der Fehlerelemente sind. Außer-

dem wird die CTA für jede Sorte $s \in S$ erweitert um eine Gleichheitsoperation, welche die Gleichheit auf Termen definiert:
eq.s : s s → bool.
Die Operation liefert true, falls zwei von error.s verschiedene Terme syntaktisch gleich sind.
Wird im folgenden von einer CTA gesprochen, so ist immer von einer CTA A´ die Rede, deren Träger entstehen aus:
$\Lambda\ s \in S\ A_s' := A_s \cup \{error.s\}$.
Für die Operationen gilt:
1. $\Lambda\ s \in S.\ \Sigma_{A'} := \Sigma_A \cup \{eq.s : s\ s \to bool,\ error.s : \to s\}$.
2. $op(t_1,\ldots,t_n)$=error.s, wenn mindestens ein t_i Error liefert. •

Theorem
Sei $\langle S,\Sigma,E\rangle$ eine Spezifikation.
Dann gibt es eine initiale (Σ,E)-Algebra, die eine CTA ist. •

So wie im letzten Abschnitt, wo die Definition eines spezifizierten ADT als die Isomorphieklasse der initialen Algebren aufgefaßt wurde, läßt sich nun die Definition eines spezifizierten ADT als die Isomorphieklasse der CTA auffassen.
Obwohl dieser Ansatz weniger abstrakt ist als der vorige, besitzt er doch die gleiche Mächtigkeit, weil zu jeder algebraischen Spezifikation eine initiale CTA existiert. Da es keinen Algorithmus gibt, der eine CTA zu einer gegebenen Spezifikation konstruieren könnte, soll mit ASPIK eine Methode vorgestellt werden, die es ermöglicht, einen Datentypen durch seine kanonischen Normalformen zu definieren.

2.2. Parametrisierung

2.2.1. Parametrisierung von Signaturen

Will man einen ADT mit den bisher vorgestellten Methoden spezifizieren, gelingt das nur unter Festlegung aller Datenbereiche und aller dazugehörenden Operationen. Daß dies aber nicht immer sinnvoll ist, zeigt folgendes Beispiel:
Es soll eine Liste oder ein Keller spezifiziert werden. In beiden Fällen sind die elementspezifischen Operationen von Interesse. Der Inhalt des Kellers oder der Liste ist an dieser Stelle nicht von Belang. Es reicht völlig aus, wenn man weiß, daß irgendwelche Elemente für den Keller oder die Liste zur Verfügung stehen. So können etwa die Operationen
first : list → elem bzw. push : elem stack → stack
ohne jegliche Annahme über elem definiert werden.
Der formale Parameter elem wäre dann im konkreten Anwendungsfall - der Instanziierung - zu aktualisieren. Ein solches Konzept erlaubt eine schrittweise Verfeinerung von Signaturteilen.
Es reicht i.a. nicht aus, Sorten als aktuelle Parameter übergeben zu können. Im allgemeinen wird der formale Parameter nicht nur aus Sorten- sondern auch aus Operationssymbolen bestehen. Bei der Aktualisierung dieser Symbole muß gefordert werden, daß formale und zugehörige aktuelle Operationen bis auf Umbenennung die gleiche Stelligkeit und Wertsorte haben. Formal heißt das, die Parameterzuordnung muß ein Signatur-Morphismus sein. Das führt zu der Frage, wann eine Aktualisierung korrekt ist.

Definition
Sei A eine (S,Σ)-Signatur. Dann seien folgende Funktionen definiert:
1. $arity: \Sigma \to S^*$ (Stelligkeit eines Operationssymbols).
   $\Lambda\ w \in S^*, s \in S.\ \Lambda\ f \in \Sigma_{w,s}.\ arity(f)=w.$
2. $sort: \Sigma \to S$ (Wertsorte eines Operationssymbols).
   $\Lambda\ w \in S^*, s \in S.\ \Lambda\ f \in \Sigma_{w,s}.\ sort(f)=s.$

Definition

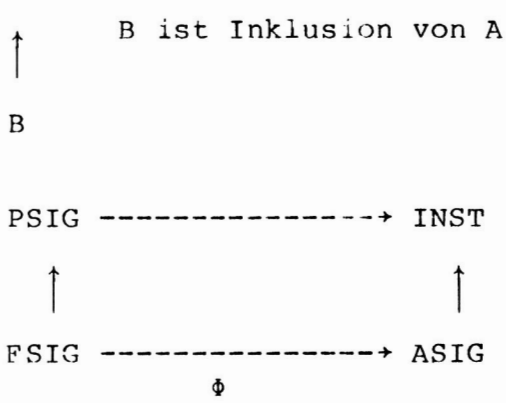
Seien $A=(S,\Sigma)$ und $B=(S',\Sigma')$ Signaturen.
Ein Signatur-Morphismus $\Phi:A \to B$ ist ein Paar von Funktionen
$\langle \Phi_S:S \to S' , \Phi_\Sigma:\Sigma \to \Sigma' \rangle$ mit:

1. $\Lambda\ f\in\Sigma.\ arity'(\Phi_\Sigma(f))=\Phi_S^*(arity(f))$.
2. $\Lambda\ f\in\Sigma.\ \Phi_S(sort'(f))=sort(\Phi_\Sigma(f))$.

(Φ_S^* sei die Fortsetzung von Φ_S auf Wörter.)

Das Ergebnis der Instanziierung ist eine Pushout-Konstruktion. Pushouts sind bis auf Isomorphie eindeutig bestimmt. Folgende Graphik veranschaulicht den Vorgang:

Notation: PSIG parametrisierte Signatur
INST Signatur der Instanz (Ergebnis der Instanziierung)
FSIG formaler Parameter
ASIG aktueller Parameter



Der eigentliche Vorgang hierbei ist folgender:
PSIG wird zu PSIG', indem der Signaturanteil von FSIG in PSIG gemäß Φ umbenannt wird. Das Ergebnis INST ist dann
PSIG' u ASIG.

2.2.2. Parametrisierung in Signatur-Hierarchien

Die Parametrisierung ist ein horizontales Strukturierungskonzept. Abstrakte Signaturteile werden auf konkrete abgebildet, wobei die Abbildung ein Signatur-Morphismus sein muß. Einen vertikalen Strukturierungsmechanismus, bei dem eine Signatur erweitert wird, bietet der hierarchische Aufbau von Signaturen. Vor allem bei großen Signaturen kann wegen der Fülle der Daten leicht die Übersicht verloren gehen. In hierarchisch angeordneten Signaturen wird auf der jeweils obersten Stufe einer Hierarchie das zu spezifizierende Problem global beschrieben. Auf einzelne kleinere Einheiten, die in sich logisch zusammenhängen, wird nicht eingegangen. Dieses Verfahren bietet neben der größeren Übersicht den Vorteil, daß die einzelnen Einheiten nur einmal zu spezifizieren sind und in mehreren Kontexten verwendet werden können.
Theoretisch gesehen wird die hierarchische Beziehung aufgebaut durch Signatur-Inklusionen. Signatur A benutzt Signatur B bedeutet, daß in A die Sortensymbole von B zur Definition der Stelligkeit von Operationssymbolen in A zur Verfügung stehen.
Die an die Instanziierung gestellte Forderung, ein Signatur-Morphismus zu sein, ist für die Aktualisierung formaler Parameter in Signatur-Hierarchien nicht ausreichend, da deren Struktur dabei nicht berücksichtigt wird. Darum ist es erforderlich, eine weitere Anforderungen an den Instanziierungsvorgang zu stellen.
Die Instanziierung einer Signatur-Hierarchie geschieht durch das schrittweise Aktualisieren der einzelnen Signatur-Anteile, wobei zuerst die unterste Signatur aktualisiert wird, dann die Signaturen, die diese benutzen usw. Daß bei dieser Vorgehensweise die ursprüngliche hierarchische Struktur verloren gehen kann, zeigt das folgende Beispiel:

Notation:	PSIG	Signatur an der Spitze der Hierarchie
	$FSIG_i$	formaler Signaturteil in der Hierarchie
	$ASIG_i$	aktueller Signaturteil
	$fsort_i$ / $asort_i$	formales / aktuelles Sortensymbol
	fop_i / aop_i	formales / aktuelles Operationssymbol

PSIG

↑

$FSIG_1$

↑ $fsort_1$

fop_1:$fsort_1$ $fsort_2$ → $fsort_2$

$FSIG_2$

$fsort_2$,$fsort_3$

fop_2:$fsort_2$ $fsort_2$ → $fsort_2$

fop2:fsort2 fsort2 → fsort2

$ASIG_1$

↑ $asort_1$,$asort_2$

aop_1:$asort_1$ $asort_3$ → $asort_2$

$ASIG_2$

$asort_3$

aop_2:$asort_3$ $asort_3$ → $asort_3$

Aktualisierung: $\Phi = \Phi_1 \cup \Phi_2$

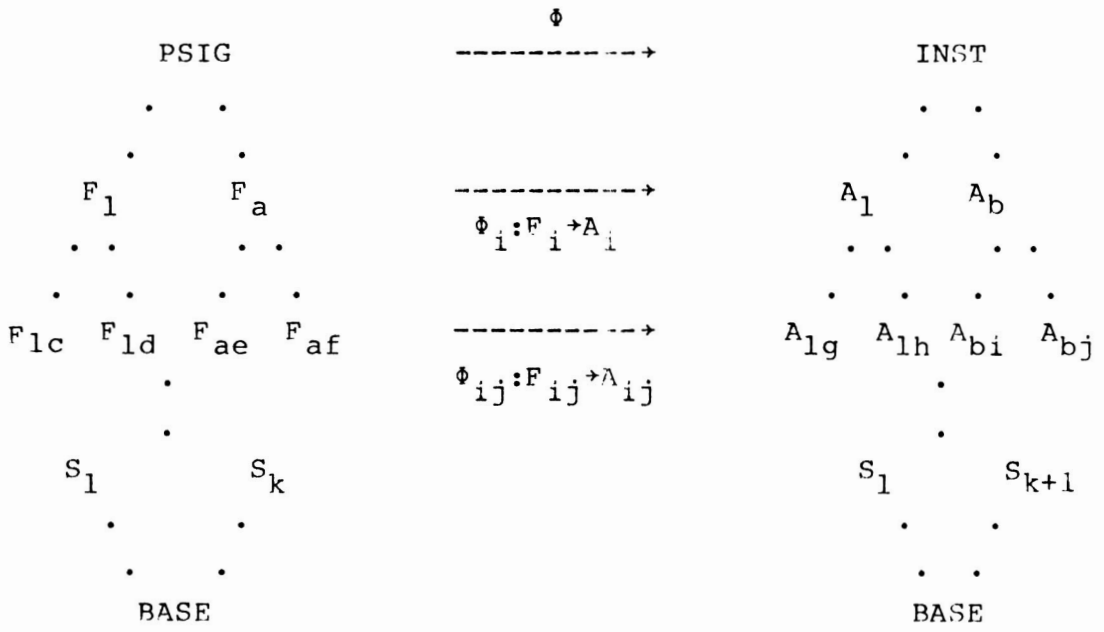
Φ_1 = $FSIG_2$ → $ASIG_1$
$fsort_2$ → $asort_3$
$fsort_3$ → $asort_2$
fop_2 → aop_2

Φ_2 = $FSIG_1$ → $ASIG_2$
$fsort_1$ → $asort_3$
fop_1 → aop_2

Bei der angegebenen Aktualisierung handelt es sich um einen Signaturmorphismus. Aber die Struktur "$FSISG_1$ benutzt $FSIG_2$" geht bei der Aktualisierung verloren, denn der aktuelle Parameter von $FSIG_2$ benutzt den aktuellen Parameter von $FSIG_1$. Die zusätzliche Forderung, die es also an eine Instanziierung einer Signatur-Hierarchie zu stellen gilt, ist die Hierarchie-Forderung. Sie besagt:
Benutzt ein formaler Parameter F_i einen formalen Parameter F_j, und F_j wurde aktualisiert durch den aktuellen Parameter A_j, so muß der aktuelle Parameter für F_i A_j benutzen, oder selbst A_j sein.

Folgende Graphik veranschaulicht die Instanziierung in Signaturhierarchien:

Notation: PSIG parametrisierte Signatur an der Spitze einer Signatur-Hierarchie.
INST Instanz.
F_i formale Parameter.
A_i aktuelle Parameter.
.. Signatur-Inklusionen.
S_i Signaturen die in den darüberliegenden gebraucht werden, aber keine formalen Parameter sind.
BASE unterste Spezifikation in der Hierarchie.



An die Aktualisierung Φ_i werden zwei Bedingungen gestellt:

1. Φ_i ist ein Signatur-Morphismus; d.h. die Aktualisierung von F_i muß mit den Aktualisierungen der von F_i benutzten Parametern verträglich sein. Weiter wird dadurch garantiert, daß alle S_i auch vom aktuellen Parameter benutzt werden, da die Signaturteile, welche nicht aktualisiert werden, identisch abgebildet werden.
2. Die Hierarchie-Forderung muß erfüllt sein; d.h. benutzt F_i F_{ij} und wurde F_{ij} durch A_{ij} aktualisiert, so muß der aktuelle Parameter A_i von F_i ebenfalls A_{ij} benutzen.

3.1. Aufbau einer ASPIK-Spezifikation

Bisher wurden zwei Ansätze vorgestellt, einen ADT zu spezifizieren, wobei sich zeigte, daß die algorithmische Spezifikationsmethode weniger abstrakt ist als die rein algebraische, da in der algebraischen Methode keine konrete Algebra angegeben wird. Um die Vorteile beider Methoden zu vereinigen, wird nun mit ASPIK eine Spezifikationssprache vorgestellt, die beide Abstraktionsebenen umfaßt und dem Benutzer größte Flexibilität in Design-Entscheidungen läßt, da

- Parameter als selbstständige Spezifikationen behandelt werden;
- es möglich ist, Hierarchien über Parametern und parametrisierten Spezifikationen zu definieren;
- parametrisierte Hierarchien partiell instanziiert werden können.

Bemerkung: Wird im folgenden von einer Spezifikation gesprochen, so ist immer der Spezifikationsanteil an der Spitze einer Hierarchie von Spezifikationen gemeint.

In ASPIK wird nach drei Typen von Spezifikationen unterschieden:
sspec einfache Spezifikation ohne Parameter;
pspec parametrisierte Spezifikation;
parm Parameter-Spezifikation.

Eine Spezifikation (hier sspec) ist folgendermaßen strukturiert:

Kopf (Interface der Spezifikation)

use sspec ...	einfache Spezifikationen	Hierarchie der Spezif.
pspec ...	parametrisierte Spezif.	
parm ...	Parameter Spezif.	
public sorts ...	neue Sorten	Signaturanteil an der CTA
ops ...	neue Operationen	
properties ...	Gleichungen	Eigenschaften der CTA-Operationen

Körper

constructors ...	Herbrand-Universum H_S	Definition der CTA Träger
auxiliaries ...	Hilfsfunktionen	
define carriers ...	Carrier $C_S \subseteq {}^cH_S u\{Error.s\}$	
define constructors ...	als Public-Operationen	Definition der CTA Operationen
private ops ...	außerhalb nicht verfügbar	
define ops ...	auf C_S	

Die Spezifikationen bestehen aus Spezifikationskopf und Spezifikationsrumpf.

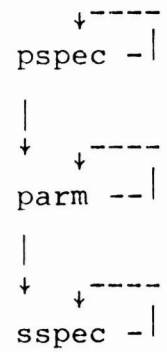
1. Der Spezifikationskopf

Der Spezifikationskopf ist die Schnittstelle zur Außenwelt. Er legt fest:

- auf welchen anderen Spezifikationen diese aufbaut;
- Namen der Sorten, Namen und Stelligkeit der Operationen und erklärt diese als zugreifbar für andere Spezifikationen;
- die Eigenschaften dieser Operationen.

1.1. Use-clause

In der use-clause wird die hierarchische Struktur beschrieben, deren Spitze die Spezifikation ist. Für die drei Spezifikations-Typen gibt es folgende use-Möglichkeiten:



Spezifikation A benutzt Spezifikation B heißt, daß in A die Vereinigung aller Signaturen aus der Hierarchie B erweitert wird um die Signatur von A. In der use-clause wird somit das downwards-interface einer Spezifikation definiert; das sind alle Sorten- und Operationssymbole die direkt oder indirekt über die use-Relation zu erreichen sind.

Die use-Relation, deren transitive Hülle keine Zyklen enthalten darf, definiert somit eine partielle Ordnung auf den benutzten Spezifikationen. Weiterhin darf die use-clause nicht leer sein bis auf eine Ausnahme: die use-clause der standardmäßig vorhandenen Spezifikation **BOOLSPEC** ist leer. Beide Forderungen zusammen

garantieren einen hierarchischen Spezifikationsaufbau und das Vorhandensein boolscher Konstanten und Operationen in jeder Spezifikation. Es besteht die Möglichkeit, in einer restrict-Klausel einzelne Operationen von benutzten Spzifikation, aus der use-Beziehung auszuschließen.

1.2. Public-clause

In dieser Klausel wird der Signatur-Anteil der Spezifikation an der CTA angegeben. Die public-Klausel zusammen mit dem downwards-interface liefert das upwards-interface, das in jeder Spezifikation, die diese benutzt, zur Verfügung steht. Die in den Operationen verwandten Sortensymbole müssen entweder in der Spezifikation deklariert worden sein, oder über das downwards-interface zu erreichen sein.

1.3. Properties

Hier werden die Eigenschaften von in der Public-clause eingeführten Operationen beschrieben, durch Implikationen folgender Form:
$e_1 \wedge \ldots \wedge e_n \Rightarrow e_{n+1}$, wobei
e_i (i=1...n) Gleichungen und e_{n+1} entweder eine Gleichung oder eine Ungleichung ist.
Die Variablen in den Implikationen sind allquantifiziert über alle nicht Error-Elemente der entsprechenden CTA-Carrier. Die Gleichheitszeichen bedeuten die syntaktische Gleichheit zwischen Termen im Gegensatz zu der für jede Sorte s implizit definierten Funktion eq.s. Für zwei Terme $t_1, t_2 \in H(\Sigma, s)$ gilt:

1. $t_1 = t_2 \quad \simeq eq.s(t_1, t_2)$ wenn $t_1, t_2 \notin \{error.s\}$
2. $t_1 = error.s \quad \simeq false$ wenn $t_1 \notin \{error.s\}$
3. $error.s = error.s \simeq true$

2.Spezifikations-Körper

Nach Festlegung des Signatur-Anteils der Spezifikation an der CTA im Spezifikations-Kopf, erfolgt für sspecs und pspecs die Definition der Trägermengen und der Operationen im Spezifikations-Körper.

2.1. Definition der CTA-Trägermengen

Die Trägermenge einer Sorte s einer CTA ist eine Teilmenge des Herbrand-Universums über deren Signatur. Zwei Fälle sind zu unterscheiden. Im einfacheren Fall sind die Träger bestimmt durch eine Teilmenge Σ' von Operationen aus der Signatur der Algebra Σ, (mit $\Sigma' \subseteq \Sigma$). Im zweiten Fall hingegen ist das Herbrand-Universum $H(\Sigma,s)$ einer Sorte s nicht isomorph zur Trägermenge der Sorte des gewünschten Datentyps. ASPIK bietet die Möglichkeit, eine Teilmenge $C_s \subseteq H(\Sigma,s)$ zu definieren, sodaß C_s Trägermenge einer Sorte s ist. Die Definition der Trägermenge erfolgt in drei Schritten:

1) Hinter dem Schlüsselwort constructors steht eine Liste von public-Operationen mit Zielsorte s für jede in dieser Spezifikation deklarierten public-Sorte. Um zu veranschaulichen, daß die Konstruktoren Elemente des Herbrand-Universums sind, tragen sie einen * als Präfix. Die Konstruktoren erzeugen das Herbrand-Universum für jede Sorte s.
2) Ist die Trägermenge der Sorte s eine echte Teilmenge des Herbrand-Universums, muß für die Sorte s ein charakteristisches Prädikat is-s definiert werden. Für diese Definition können auxiliaries deklariert und definiert werden. Auxiliaries sind Hilfsfunktionen über dem Herbrand-Universum, mit denen das charakteristische Prädikat definiert wird. Auxiliaries haben ein $ als Präfix.
3) Definiert wird das charakteristische Prädikat is-s in der define carrier Klausel. Es bestimmt die Trägermenge C_s einer Sorte s durch:
   $C_s=\{t \mid t\in H(\Sigma',s) \wedge \text{is-s}(t)=\text{true}\} \cup \{\text{error}\}$.
   is-s heißt auch Akzeptor-Funktion der Sorte s.

Die Klauseln auxiliaries, define auxiliaries und define carriers sind optional. Fehlen sie, gilt das triviale Prädikat, das stets true liefert.

2.2. Definition der CTA-Operationen

Jede public-Operation muß auf den entsprechenden Trägern definiert werden. Die Definitionen müssen wegen der CTA-Anforderung

die Konstruktor-Eigenschaft erfüllen, was dadurch erzwungen wird, daß die Definition der Operationen in zwei Schritten erfolgt. Im ersten werden alle Konstruktoren, die zur Definition der Trägermengen benutzt werden, als public-Operationen definiert. Im zweiten Schritt werden die restlichen public-Operationen definiert, wobei private Operationen (versteckte Operationen) verwendet werden können. Die linke Seite einer Operationsdefinition besteht aus dem Operationsaufruf mit Variablen der Sorten entsprechend der Stelligkeit der Operationen. Die rechte Seite ist ein Operationsschema über den Variablen der linken Seite. Folgende Schemata können zur Definition der Operationen verwendet werden:

- eine 0-stellige Operation;
- die Konstante error.s; diese liefert das error-Element der Sorte s;
- eine Folge von Operationsaufrufen, wobei auch rekursive Operationsaufrufe erlaubt sind;
- ein if-then-else Schema, wobei then- und else-Teil wieder Operationsschemata sein können;
- ein case-Schema, das sich auf die Tatsache bezieht, daß die Argumente der Operationen Elemente der Trägermenge sind; abhängig vom äußersten Konstruktor wird ein Operationsschema angegeben.
- ein let-Schema, bei dem eine neue Variable als Abkürzung für einen Term eingeführt werden darf.

Private-Operationen können außerhalb der Spezifikation nicht verwendet werden, da sie nicht im Spezifikationskopf deklariert werden. Alle auxiliaries sind automatisch als private-Operationen verfügbar. Das Präfix $ wird weggelassen. Der Unterschied zwischen private-Operationen und auxiliaries liegt darin, daß die letzteren über dem Herbrand-Universum, private-Operationen aber über den Trägermengen definiert sind.

3.2. Parametrisierung in ASPIK

In ASPIK sind formale Parameter (gekennzeichnet durch parm) eigenständige Spezifikationen. Das bietet den Vorteil, daß sie zum einen wie einfache Spezifikationen hierarchisch aufgebaut werden können und zum anderen, daß verschiedene andere Spezifikationen diese benutzen können. In vier Punkten unterscheidet sich eine Parameter-Spezifikation von einer einfachen:

1. Ein parm hat keinen Spezifikations-Körper, da seine Sorten und Operationen im aktuellen Parameter definiert werden;
2. Die Properties eines parms stellen Anforderungen an den aktuellen Parameter dar; sie müssen daher im aktuellen Parameters gelten.
3. Ein parm darf auf anderen parms aufgebaut sein; darum darf er sowohl parms, als auch sspecs benutzen.
4. Die restrict-Möglichkeiten in der use-Klausel entfallen.

Parametrisierte Spezifikationen (gekennzeichnet durch pspec) unterscheiden sich in zwei Punkten von einfachen:

1. Sie benutzen wenigstens einen parm.
2. Sie dürfen ihrerseits wieder pspecs benutzen.

Pspecs und parms benutzen parms über parm-Variablen, so daß also die Möglichkeit besteht, denselben parm in einer Spezifikation mehrfach zu verwenden und die verschieden parm-Variablen des selben Typs verschieden zu aktualisieren. Parm-Variablen beginnen mit # und werden in der parm-Deklaration eingeführt.

Parms werden nach zwei Arten von einander unterschieden:

1. Elementare parms benutzen keine anderen parms; es fehlt folglich die parm-Deklaration.
2. Zusammengesetzte parms (mit parm-Deklaration) benutzen andere parms.

Da pspecs immer wenigstens einen parm benutzen, besitzen sie somit auch immer eine Parameter-Deklaration.

Parm-Variablen sind immer lokal bzgl. einer Spezifikation. Dieses wird klar an einem Beispiel:
Ordelem(#e:Elem) sei ein zusammengesetzter parm, in dem eine Ordnung auf Elementen festgelegt wird. Der elementare parm Elem stelle die Elemente dar. Eine Variable vom Typ Ordelem kann dann immer nur zusammen mit einer Variablen vom Typ Elem auftreten, da #e außerhalb von Ordelem nicht zugreifbar ist, also:
#oe:Ordelem(#el:Elem).
Dagegen wären #oe:Ordelem oder #oe:Ordelem(#el,#e2:Elem) unzulässig. Hieraus ergibt sich die Frage, wann eine parm-Variable wohl definiert ist.
Bevor hierauf eine Antwort gegeben wird, sei auf zwei Arten von parm-Variablen hingewiesen:

1. direct-parm-Variablen, die auf dem Top-Level einer parm-Deklaration definiert wurden;
2. component-parm-Variablen, die auf einem tieferen Level definiert wurden.

Definition

parm-declaration :: (varid$^+$: parm-term)$^+$
parm-term :: (specid [(component-parm-declaration)]
pspec-term :: specid(varid$^+$)
component-parm-declaration
:: (varid$^+$ [: parm-term])$^+$ •

Zum ersten Auftreten einer parm-Variablen gehört immer die Definition durch einen mit parm-term, jedes weitere Auftreten der Variablen in der parm-Deklaration muß davon abgeleitet sein (ohne parm-term). Der Typ einer parm-Variablen ist gegeben durch den entsprechenden parm-term; sie ist wohl definiert, wenn ihr Typ wohldefiniert ist.

Definition

Gegeben sei ein parm-term PT bzgl. Spezifikation P. Dieser werde erweitert zu $PT_e := P(\#x_1:T_{el},\ldots,\#x_n:t_{en})$ durch iteratives Ersetzen aller abgeleiteten parm-Variablen durch deren Definition.

In der gleichen Weise werde die parm-Deklaration von P erweitert zu $P_e := (\#y_1:T_{e1},\ldots,\#y_n:T_{en})$.
PT ist wohldefiniert, wenn eine Abbildung σ von P's lokalen Variablen in $\#x_1,\ldots,\#x_n$ und deren component-Variablen existiert, sodaß $\sigma_e(P_e)=\sigma_e(PT_e)$, wobei σ_e auf einen parm-term angewandt, jede Variable #y durch $\sigma(\#y)$ ersetzt. •

Theorem
Wenn σ existiert, so ist es eindeutig bestimmt.

Notation: σ_{PT} oder $\sigma_{\#v}$ (falls #v:PT(...)).
σ_{PT} ($\sigma_{\#v}$) heißt reassignment map von PT (#v).

Pspecs benutzen andere pspecs über pspec-terms. In gleicher Weise wie bei parm-terms stellt sich die Frage, wann ein gegebener pspec-Term wohldefiniert ist. Analog zum Vorgehen beim parm-term werden ein pspec-term PST und die Parameter-Deklaratin PD der entsprechenden pspec erweitert. Der Term ist wiederum wohl definiert, wenn es eine Abbildung σ von den lokalen Variablen von PD in die Variablen von PST gibt, sodaß $\sigma(PD_e)=\sigma_e(PST_e)$.
σ_{PST} heißt reassignment map von PST.
Nach Erörterung der use-Beziehungen von pspecs und parms gilt es nun zu klären, was an Sorten- und Operationssymbolen einer Spezifikation über diese zur Verfügung steht. Wie schon erwähnt wird unterschieden zwischen dem upwards- und downwards-interface einer Spezifikation. Beides sei im folgenden definiert.
Ein Interface ist eine indizierte Menge von Signaturen. Eine Signatur $\langle S,\Sigma\rangle_{SPEC}$ aus dem downwards-interface bedeutet, daß die Spezifikation direkt oder indirekt SPEC mit deren Sorten S und einer Teilmenge (gemäß der restict-Klausel) der public-Operationen Σ benutzt. Diese benutzten Sorten und Operationssymbole dürfen in der Spezifikation immer mit Präfix SPEC verwendet werden. Sie müssen mit Präfix verwendet werden, wenn sie mehrdeutig sind, d.h. mehrere Signaturen aus dem Interface enthalten das gleiche Sorten- bzw. Operationssymbol.

Definition

Das **downwards-interface** einer Spezifikation erhält man durch:

1. Definiere für jede direkte Parameter-Variable #v deren upwards -interface.
   Definiere für jeden pspec-term aus der use-Klausel dessen upwards-interface.
2. Verschmelze die Interfaces aus 1. mit den upwards-interfaces aller sspecs aus der use-Klausel durch indexweises Vereinigen der Signaturen.
3. Für jeden Index I mit Restriktionen in der use-Klausel, lösche in I's Signatur alle Operationen, bis auf die in der Klausel aufgeführten. •

Definition

Das **upwards-interface einer sspec** SSP ist definiert durch ihr downwards-interface vereinigt mit der public-clause von SSP. •

Definition

Das **upwards-interface einer parm-Variable** #v:(P...) leitet sich ab von P's upwards-interface durch Ersetzung jedes Parameter-Variablen-Index #w durch das reassignment $\sigma_{\#v}(\#w)$ und Ersetzung von von Index P durch #v. •

Definition

Das **upwards-interface eines pspec-term** $PS(\#x_1,...,\#x_n)$ leitet sich ab von PS's upwards-interface durch Ersetzung jedes Parameter-Variablen-Index #w durch das reassignment:

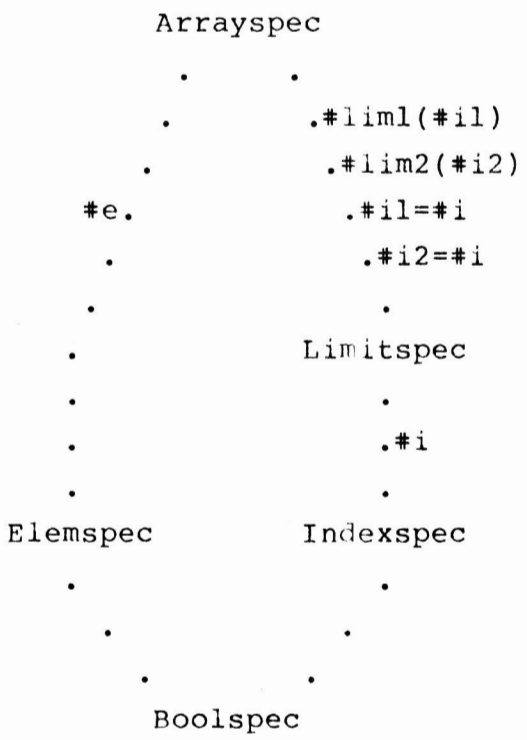
$\sigma_{PS(\#x1,...,\#xn)}(\#w)$ $i=1...m.\ \Lambda\ \#w_i \varepsilon PS'.$,

durch Ersetzung jedes Pspec-Term-Index PS'(#w1,...,#wm) durch das Anwenden des reassignments:

$\sigma_{PS(\#x1,...,\#xn)}(\#w_i)$ $i=1...m.\ \Lambda\ \#w_i \varepsilon PS'.$

und durch die Ersetzung des Index PS durch PS(#x1,...,#xn). •

In der Praxis sollen Spezifikations-Hierarchien mit Hilfe des interaktiven Systems SPESY am Rechner erstellt werden. Auch die Instanziierung, bei der eine Hierarchie mit einer pspec oder einem parm an der Spitze durch eine andere Hierarchie mit entsprechenden aktuellen Parametern aktualisiert wird, läuft automatisch in Abhängigkeit des SPESY-Kommandos instantiate ab. Darum sei im folgenden dieses Kommando beschrieben. Zur Veranschaulichung diene ein zweidimensionales Array mit variablen Grenzen und folgendem hierarchischen Aufbau:



Die Hierarchie besteht aus folgenden Spezifikationen:

```
sspec Boolspec

   public sorts bool
          ops true,false: → bool
              not: bool → bool
              and,or: bool bool → bool

   properties   not(true) = false
                not(false) = true
                not(and(x,y)) = or(not(x),not(y))

   constructors *true, *false

   define ops
      not(x) :=
                case x is
                   *true : false
                   *false : true
                esac

      and(x,y) :=
                case x is
                   *true : y
                   *false : false
                esac

      or(x,y) :=
                case x is
                   *true : true
                   *false : y
                esac

endspec
```

```
sspec Natspec

   use sspecs   Boolspec

   public sorts nat
            ops 0: → nat
                succ: nat → nat
                plus: nat nat → nat
                max: nat nat → nat
                lt: nat nat → bool

   properties   plus(succ(x),y) = succ(plus(x,y))
                plus(x,y) = plus(y,x)
                lt(x,x) = false
                lt(x,succ(x)) =true
                lt(x,y) = false & lt(y,x) = false ==> eq.nat(x,y)
                                                      = true

   constructors *0, *succ

   define ops
      plus(x,y) :=
                case x is
                   *0 : y
                   *succ(xl) : succ(plus(xl,y))
                esac

      lt(x,y) :=
                case x is
                   *0 : not(eq.nat(x,y))
                   *succ(xl) : case y is
                                  *0 : false
                                  *succ(yl) : lt(xl,x2)
                               esac
                esac

      max(x,y) :=
                if lt(x,y) then y else x

endspec
```

```
parm Elemspec

   use sspecs    Boolspec

   public sorts elem

endspec


parm Indexspec

   use sspecs    Boolspec

   public sorts index
            ops suc: index → index
                less: index index → bool
                max: index index → index
                min: → index

   properties   less(x,suc(x)) = true
                less(x,x) = false
                less(min,suc(x)) = true
                less(x,y) = true ==> not(less(y,x)) = true

endspec


parm Limitspec(#i:Indexspec)

   use parms     #i

   public ops    limit: → index

   properties   lt(limit,suc**20(min)) = true
                lt(min,limit) = true

endspec
```

```
pspec Arrayspec (#e:Elemspec,
                 #liml:Limitspec(#indl:Indexspec),
                 #lim2:Limitspec(#ind2:Indexspec))

   use parms     #e,#liml,#lim2

   public sorts array
          ops new: → array
              put: #indl.index #ind2.index elem array → array
              get: #indl.index #ind2.index array → elem
              isdef: #indl.index #ind2.index array → bool
              isnew: array → bool

   properties

   constructors *new, *put

   auxiliaries  $boundl: array → #indl.index
                $bound2: array → #ind2.index

   define auxiliaries
      $boundl(a) :=
                case a is
                   *new : #indl.min
                   *put(m,n,e,al) : m
                esac

      $bound2(a) :=
                case a is
                   *new : #ind2.min
                   *put(m,n,e,al) : #ind2.max(n,$bound2(al))
                esac
```

```
define carriers
   is-array(a) :=
              case a is
                 *new : true
                 *put(m,n,e,al) :
                    if not(is-array(al))
                    then false
                    else and(or(#indl.less($boundl(al),m),
                                and(eq.#indl.index(m,$boundl(al)),
                                    #ind2.less($bound2(al),n))),
                             and(#indl.less(m,#iml.limit),
                                 #ind2.less(n,#lim2.limit)))
              esac

define constructors
   new := *new

   put(m,n,e,al) :=
              if and(or(#indl.less($boundl(al),m),
                        and(eq.#indl.index(m,$boundl(al)),
                            #ind2.less($bound2(al),n))),
                     and(#indl.less(m,#iml.limit),
                         #ind2.less(n,#lim2.limit)))
              then *put(m,n,e,al)
              elsif not(and(#indl.less(m,#iml.limit),
                            #ind2.less(n,#lim2.limit)))
              then error.array
              else case al is
                      *new : error.array
                      *put(ml,nl,el,a2) :
                           put(ml,nl,el(put(m,n,e,a2)))
                   esac
```

```
   define ops
      get(m,n,al) :=
                case al is
                   *new : error.elem
                   *put(ml,nl,el,a2) :
                    if and(eq.#indl.index(m,ml),eq.#ind2.index(n,nl))
                    then el
                    elseif or(#indl.less(m,ml),
                                  and(eq.#indl.index(ml,m),
                                      #ind2.less(n,nl)))
                    then get(m,n,a2)
                    else error.array
                esac

      isdef(m,n,al) :=
                case al is
                   *new : error.bool
                   *put(ml,nl,e,a2) :
                    if and(eq.#indl.index(m,ml),eq.#ind2.index(n,nl))
                    then true
                    elseif or(#indl.less(m,ml),
                                  and(eq.#indl.index(ml,m),
                                      #ind2.less(n,nl)))
                    then isdef(m,n,a2)
                    else error.bool
                esac

      isnew(al) :=
                case al is
                   *new : true
                   *put(ml,nl,e,a2) : false
                esac

endspec
```

Das Instanziierungs-Kommando besteht aus vier Teilen. Nach Aufruf des Kommandos besteht im zweiten Teil die Möglichkeit, die Variablen zu deklarieren, die für die Aktualisierung formaler Parameter im dritten Teil benötigt werden. Zuletzt besteht die Möglichkeit, Sorten- oder Operationssymbolen neue Namen zu geben.

Definition

instantiate-Kommando:
instantiate specid to specid
declare [(parm-declaration)]
actualize [actualization-clause^{+}]
rename [rename-clause]

actualization-clause ::
    formal^{+} by match   actual |
    formal^{+} by instance actual |
    formal^{+} by   actual [rename-clause]

formal ::
    varid | pspec-term

actual ::
    formal | specid

rename-clause ::
    [sorts (sortid by sortid)*]
    [ops   (opid   by opid)*]

1. instantiate specalt to specneu

Der Aufruf des Instanziierungs-Kommandos erfolgt über das Schlüsselwort instantiate, gefolgt von einem pspec- oder parm-Namen, hier specalt, der eine Spezifikation an der Spitze einer Hierarchie von Parametern oder parametrisierten Spezifikationen benennen muß. Um diese Forderung zu garantieren, müssen zum einen die

Parameter-Deklaration von specalt wohl definiert sein bzgl. der existierenden Spezifikationen, zum anderen muß die use-Klausel von specalt korrekt sein, d.h. in der use-Beziehung dürfen keine Zyklen auftreten, und alle dort aufgeführten Spezifikationen müssen definiert sein. Dem Schlüsselwort to folgt ein Spezifikationsname (hier specneu), welcher im bisherigen System nicht definiert sein darf, denn er bezeichnet die Spezifikation an der Spitze der zu generierenden Hierarchie.

2. declare actual-parm-declaration

Dem Schlüsselwort declare folgt die aktuelle Parameter-Deklaration. Diese muß wohl definiert sein bzgl. aller existierenden Spezifikationen. Formale Parameter können ihrerseits wieder durch pspecs oder parms aktualisiert werden. Da die Parameter-Variablen in pspecs und zusammengesetzten parms lokal bzgl. der jeweiligen Spezifikation sind, können an dieser Stelle die für die aktuellen parametrisierten Parameter benötigten parm-Variablen eingeführt werden. Um Eindeutigkeit zu garantieren, müssen die eingeführten Variablen verschieden sein von allen Variablen aus der Parameter-Deklaration von specalt. Im Verlauf der Aktualisierungen müssen alle direkten aktuellen Parameter-Variablen benutzt werden, denn sonst wären sie nutzlos. Werden nur sspecs als aktuelle Parameter benutzt, entfällt diese Klausel.

3. actualize actualization

Die Angabe des Signatur-Morphismus, durch den formalen Parametern aktuelle zugewiesen werden, folgt dem Schlüsselwort actualize. Die Aktualisierung kann in mehreren aufeinander folgenden Schritten erfolgen. Unter der Annahme, daß die Aktualisierungen 1...i-1 der formalen Parameter f1,...,fk-1 korrekt verliefen, hat die Aktualisierung i folgende Form:

actualization$_i$:=

fk,...,fn by | by match | by instance ai
[sorts fsl by asl
.
.
fsm by asm]
[ops fol by aol
.
.
fop by aop]

Annmerkung: Die beiden letzten optionalen Klauseln entfallen bei den Aktualisierungsarten by_match und by_instance immer.

Für die formalen Parameter fk,...,fn sind nur zwei Möglichkeiten zugelassen:

F1: fk,...,fn sind Parameter-Variablen des selben Typs aus der Parameter-Deklaration von specalt. Daraus folgt, daß die definierenden parm-terms alle mit denselben parm-Namen (z.B. mit FP) beginnen müssen.

Definition

faε{fk,...,fn}.

1. τ_{fa}: FP → fa sei die reassignment map von FP bzgl. fa.
2. τ_{fa}(FP) := fa.

F2: fk,...,fn sind pspec-Terme. Sie bezeichnen pspecs, die entweder direkt oder indirekt von specalt benutzt werden, und die alle mit dem selben pspec-Namen beginnen (z.B. mit FP).

Definition

faε{fk,...,fn}.

1. τ_{fa}: FP → fa sei die reassignment map von FP bzgl. fa.
2. τ_{fa}(FP) := fa.

Folgende Anforderungen werden an die Aktualisierung von fk...fn gestellt:

1. Die Parameter fk,...,fn müssen verschieden sein von den zuvor aktualisierten Parametern fl,...,fk-1, da formale Parameter nur einem aktuellen Parameter zugeordnet werden können.
2. Die Aktualisierungen müssen der use-Beziehung folgen, d.h. für jedes Paar fa,fb mit a<k<b gilt:
   fa darf fb nicht benutzen.
   Diese Forderung garantiert, daß für jeden formalen Parameter, der aktualisiert wird, dessen eigene formalen Anteile bereits aktualisiert wurden.
3. Die Aktualisierung muß direkt sein. Wird ein formaler Parameter fa∈{fk,...,fn} von einem anderen fm benutzt, so muß fm in den Aktualisierungen i+1...n aktualisiert werden.
4. Die Aktualisierung darf partiell sein, d.h. es müssen nicht alle parm-Variablen bzw. pspec-Terme von specalt aktualisiert werden.

Als aktuelle Parameter können sspecs, pspecs oder parms verwandt werden. Drei Fälle können auftreten:

A1: ai bezeichnet eine sspec, z.B. AP, die im existierenden System definiert ist.

Definition

τ_{ai}: AP → ai sei die identische Abbildung.

A2: ai ist eine aktuelle Parameter-Variable #v, die unter declare eingeführt wurde, z.B. durch #v:AP(avars), wobei avars die component-parm-Variablen von #v sind.

Definition

1. τ_{ai}: AP → ai sei die ressignment map von AP bzgl. ai.
2. τ_{ai}(AP) := #v

A3: ai ist ein pspec-term, z.B. AP(avars). Seine component-parm-Variablen avars müssen unter declare eingeführt worden sein, und der Term muß bzgl. des bestehenden Systems wohl definiert sein.

Definition

1. τ_{ai}: AP → ai sei die reassignment map von AP bzgl. ai.
2. $\tau_{ai}(AP) := ai$.
3. Für alle von ai benutzten pspec-terms PT(vars).
   $\tau(PT(vars)) := PT(\sigma_{ai}(vars))$.

Der hierarchische Aufbau von Spezifikationen, und die Hierarchie innerhalb der use-Möglichkeiten der verschiedenen Spezifikationstypen stellen zwei Forderungen an ai:

1. Das Hierarchie-Requirement muß erfüllt sein.
2. Benutzt einer der formalen Parameter aus fk,...,fn eine Variable bzw. einen pspec-term fa, und fa wurde durch eine Variable (pspec-term) aktualisiert, so muß der aktuelle Parameter für fk,...,fn ebenfalls eine Variable (pspec-term) sein.

3.1. Aktualisierung by_match

Bei dieser Aktualisierungsart wird keine explizite Abbildung von Sorten und Operationen des formalen Parameters in die des aktuellen Parameters angegeben. Sie werden statt dessen automatisch den entsprechenden Sorten und Operationen eines aktuellen Parameters des selben Typs zugewiesen.
Im Fall F1: muß A2:, in Fall F2: muß A3: Anwendung finden.
In beiden Fällen muß der Typ des aktuellen Parameters AP derselbe sein, wie der der formalen Parameter FP, also AP = FP. Weiter muß es für jeden formalen Parameter fa eine Abbildung ϕ_{fa}: fa → ai geben, sodaß $\tau_{ai} = \phi_{fa} \circ \tau_{fa}$.
Wenn ϕ_{fa} existiert, so ist es eindeutig und definiert, auf welche Weise Sorten, Operationen, Variablen und pspec-Terme, die von fa benutzt werden, zu den entsprechenden von ai stehen.
Außerdem wird all das, was von fa benutzt wird, implizit über ϕ_{fa}

aktualisiert. Wurde eine der Variablen oder Terme bereits während der Aktualisierungen 1...i-1 aktualisiert, so muß die frühere Aktualisierung mit ϕ_{fa} übereinstimmen.

Beispiel 1

Das oben spezifizierte Array soll instanziiert werden zu einem zweidimensionalen quadratischen Array. Dazu sind die beiden Indizes #i1 und #i2, und die Limiten #lim1 und #lim2 miteinander zu identifizieren.

```
instantiate Arrayspec to Squarearray
declare (#lm:Limitspec(#in:Indexspec))
actualize #lim1,#lim2 by_match #lm
```

Um zu überprüfen, ob die Aktualisierung den Anforderungen entspricht, gilt es die Abbildungen $\phi_{\#lim1}$ und $\phi_{\#lim2}$ zu bestimmen:

1. Bestimmung von $\phi_{\#lim1}$:
   a. Bestimmung von $\tau_{\#lim1}$:
      #i → #ind1
      Limitspec → #lim1
   b. Bestimmung von $\tau_{\#lm}$:
      #i → #in
      Limitspec → #lm
   c. Bestimmung von $\phi_{\#lim1}$:
      #il → #in
      #lim1 → #lm
2. Bestimmung von $\phi_{\#lim2}$:
   a. Bestimmung von $\tau_{\#lim2}$:
      #i → #ind2
      Limitspec → #lim2
   b. Bestimmung von $\tau_{\#lm}$:
      #i → #in
      Limitspec → #lm
   c. Bestimmung von $\phi_{\#lim2}$:
      #i2 → #in
      #lim2 → #lm

In beiden Fällen ist ϕ eine Abbildung und Inkonsistenzen zu vorherigen Aktualisierungen gibt es keine. Die Aktualisierung ist somit korrekt und Squarearray hat folgenden Spezifikations-Kopf (der Spezifikations-Körper ist identisch zu dem von Arrayspec, mit folgenden Ausnahmen: in allen Operationsnamen sind die Teile #liml, #lim2 durch #lm und #indl, #ind2 durch #in ersetzt):

```
pspec Sqarearray (#e:Elemspec,
                  #lm:Limitspec(#in:Indexspec))

   use parms     #e,#lm,

   public sorts array
            ops new: → array
                put: #in.index #in.index elem array → array
                get: #in.index #in.index array → elem
                isdef: #in.index #in.index array → bool
                isnew: array → bool

   properties

   instantiates Arrayspec (#e:Elemspec,
                           #liml:Limitspec(#indl:Indexspec),
                           #lim2:Limitspec(#ind2:Indexspec))
   actualizes #liml,#lim2 by_match #lm

   constructors ....

   endspec
```

Anmerkung: Die neue Spezifikation erhält als Dokumentation der Instanziierung eine besondere instantiates-Klausel. Diese wird weiter unten beschrieben. •

Beispiel 2

Gegeben sei die Spezifikation Arrayspec und folgende Instanziierung:

instantiate Arrayspec to Arraynew
declare (#lm:Limitspec(#inl:Indexspec),#in2:Indexspec)
actualize #indl by match #in2
#liml,#lim2 by match #lm

Die Bestimmung von $\phi_{\#indl}$ in $Aktualisierung_1$ ergibt:

#indl → #in2

Die Bestimmung von $\phi_{\#liml}$ in $Aktualisierung_2$ ergibt:

#indl → #inl
#liml → #lm

Die Bestimmung von $\phi_{\#lim2}$ in $Aktualisierung_2$ ergibt:

#i2 → #inl
#lim2 → #lm

Die beiden Aktualisierungen sind nicht mit einander verträglich, da #indl sowohl #inl als auch #in2 zugeordnet wird. Die Instanziierung ist damit nicht korrekt. •

Beispiel 3

Gegeben:

```
pspec Arrayl (#e:Elemspec,#lim:Limitspec(#i:Indexspec))
use pspecs Arrayspec(#e,#lim,#lim)
                        .
                        .
                        .
endspec


instantiate Arrayl to Array2
declare (#el:Elemspec,
         #lml:Limitspec(#in:Indexspec),
         #lm2:Limitspec(#in))
actualize Arrayspec(#e,#lim,#lim) by match
                                      Arrayspec(#el,#lml,#lm2)
```

Es gilt wiederum die Abbildung $\phi_{Arrayspec(\#e,\#liml,\#lim2)}$ zu bestimmen:

a. Bestimmung von $\tau_{Arrayspec(\#e,\#lim,\#lim)}$:

```
#e → #e
#liml → #lim
#lim2 → #lim
#indl → #i
#ind2 → #i
Arrayspec → Arrayspec(#e,#lim,#lim)
```

b. Bestimmung von $\tau_{Arrayspec(\#el,\#lml,\#lm2)}$:

```
#e → #el
#liml → #lml
#lim2 → #lm2
#indl → #in
#ind2 → #in
Arrayspec → Arrayspec(#el,#lml,#lm2)
```

c. Bestimmung von $\phi_{Arrayspec(\#e,\#lim,\#lim)}$:

#e → #el

#lim → #lm1

#lim → #lm2

#i → #in

Arrayspec(#e,#lim,#lim) → Arrayspec(#el,#lm1,#lm2)

Die Aktualisierung definiert keine Abbildung ϕ und ist aus diesem Grund nicht korrekt. •

3.2. Explizite Aktualisierung

Im Gegensatz zu der vorherigen, wird bei dieser Aktualisierungsart eine Abbildung von den Sorten und Operationen der formalen Parameter fa∈{fk...,fn} in die des aktuellen Parameters ai explizit angegeben. Die Angabe erfolgt in der rename-Klausel. Die Symbole fsl,...,fsm müssen genau die public-Sorten, und fol,...,fop die public-Operationen von FP sein. Die aktuellen Sorten und Operationen müssen dem upwards-interface des aktuellen Parameters AP entstammen. Für jedes fa ergibt diese explizite Aktualisierung zusammen mit den vorangegangenen Aktualisierungen von parm-Variablen und pspec-Termen, die von fa benutzt werden und in früheren Aktualisierungen aktuellen Parametern zugewiesen wurden, die Abbildung ϕ_{fa}: fa → ai. Alle Sorten und Operationen der von fa benutzten sspecs werden identisch auf sich selbst abgebildet. Da die Aktualisierung ein Signatur-Morphismus sein muß, muß für jede public-Operation foj von fa, die auf die aktuelle Operation aoj abgebildet wird, gelten:
$\phi_{fa}(foj) = aoj : \phi_{fa}(arity_{foj}) \dashrightarrow \phi_{fa}(sort_{foj})$.

Beispiel 4

Arrayspec soll instanziiert werden zu einem Array, dessen Elemente wiederum aus Arrays bestehen.

```
instantiate Arrayspec to Array-array
declare    (#el:Elemspec,
            #liml1:Limitspec(#indl1:Indexspec),
            #lim2l:Limitspec(#ind2l:Indexspec))
actualize #e by Arrayspec(#el,#liml1,#lim2l)
    sorts elem by array
```

Das Ergebnis der Instanziierung ist folgende Spezifikation:

```
pspec Array-array (#el:Elemspec,
                   #liml:Limitspec(#indl:Indexspec),
                   #lim2:Limitspec(#ind2:Indexspec),
                   #liml1:Limitspec(#indl1:Indexspec),
                   #lim2l:Limitspec(#ind2l:Indexspec))

   use pspecs   Arrayspec(#el,#liml1,#lim2l)

   public sorts array
            ops new: → array
                put: #indl.index #ind2.index
                       Arrayspec(#el,#liml1,#lim2l).array array → array
                get: #indl.index #ind2.index array →
                       Arrayspec(#el,#liml1,#lim2l).array
                isdef: #indl.index #ind2.index array → bool
                isnew: array → bool

   properties
```

```
instantiates Arrayspec
             (#e:Elemspec,
              #liml:Limitspec(#indl:Indexspec),
              #lim2:Limitspec(#ind2:Indexspec))
actualizes #e by Arrayspec(#el,#liml1,#lim21)
     sorts elem by array

constructors *new, *put

auxiliaries  $boundl: array → indl.index
             $bound2: array → ind2.index

define auxiliaries
   $boundl(a) :=
             case a is
                *new : #indl.min
                *put(m,n,e,al) : m
             esac

   $bound2(a) :=
             case a is
                *new : #ind2.min
                *put(m,n,e,al) : #ind2.max(n,$bound2(al))
             esac
```

```
define carriers
   is-array(a) :=
            case a is
               *new : true
               *put(m,n,e,al) :
                  if not is-array(al)
                  then false
                  else and(or(#indl.less($boundl(al),m),
                              and(eq.#indl.index(m,$boundl(al)),
                                  #ind2.less($bound2(al),n))),
                           and(#indl.less(m,#liml.limit),
                               #ind2.less(n,#lim2.limit)))
            esac

define constructors
   new := *new

   put(m,n,e,al) :=
            if and(or(#indl.less($boundl(al),m),
                      and(eq.#indl.index(m,$boundl(al)),
                          #ind2.less($bound2(al),n))),
                   and(#indl.less(m,#liml.limit),
                       #ind2.less(n,#lim2.limit)))
            then *put(m,n,e,al)
            elsif not(and(#indl.less(m,#liml.limit),
                          #ind2.less(n,#lim2.limit)))
            then error.array
            else case al is
                    *new : error.array
                    *put(ml,nl,el,a2) :
                         put(ml,nl,el(put(m,n,e,a2)))
                 esac
```

```
define ops
   get(m,n,al) :=
              case al is
                 *new : error.Arrayspec(#el,#liml1,#lim2l).array
                 *put(ml,nl,el,a2) :
                  if and(eq.#indl.index(m,ml),eq.#ind2.index(n,nl))
                  then el
                  elseif or(#indl.less(m,ml),
                                 and(eq.#indl.index(ml,m),
                                     #ind2.less(n,nl))
                  then get(m,n,a2)
                  else error.array
              esac

   isdef(m,n,al) :=
              case al is
                 *new : error.bool
                 *put(ml,nl,e,a2) :
                  if and(eq.#indl.index(m,ml),eq.#ind2.index(n,nl))
                  then true
                  elseif or(#indl.less(m,ml),
                                 and(eq.#indl.index(ml,m),
                                     #ind2.less(n,nl))
                  then isdef(m,n,a2)
                  else error.bool
              esac

   isnew(al) :=
              case al is
                 *new : true
                 *put(m,n,e,a2) : false
              esac

endspec
```

Anmerkung

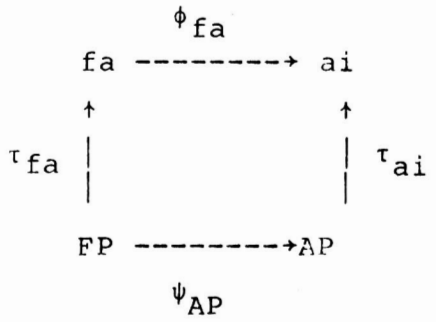
Beim letzten Beispiel ist zu beachten, daß die Sorte array durch die Zuordnung von elem → array mehrdeutig wurde bzgl. des upwards-interface von Array-array. Darum muß zum Beispiel die aktualisierte Zielsorte array in der Operation get mit dem pspec-term als Präfix versehen werden, aus dessen public-clause die Sorte stammt. Diese Qualifikation muß ebenso für alle error- und eq-Elemente im Spezifikations-Körper durchgeführt werden. Die Sorte array ohne Präfix dagegen stammt aus der public-clause von Array-array. •

3.3. Aktualisierung by_instance

Definition

instantiation-clause:
instantiates specid (parm-declaration)
[actualizes actualisation-clause^{+}]
[renames rename-clause] •

Jede Instanz einer Spezifikation AP erhält als Dokumentation der Instanziierung eine dem ausgeführten instantiate-Kommando entsprechende instantiation-clause, in der unter anderem der Signatur-Morphismus gespeichert wird. Die Instanz kann ihrerseits wieder als aktueller Parameter für einen formalen benutzt werden. Gilt, daß der Typ AP des aktuellen Parameters ai eine Instanz des Typs FP des formalen Parameters fa ist, so definiert die instantiates-clause eine Abbildung ψ_{AP}: FP → AP, und es ergibt sich folgendes Diagramm:



Das bedeutet, daß es für jeden formalen Parameter fa eine Abbildung ϕ_{fa}: fa → ai gibt, sodaß $\phi_{fa} \circ \tau_{fa} = \tau_{ai} \circ \psi_{AP}$.
Wenn das Diagramm kommutiert, ist ϕ_{fa} eindeutig bestimmt. ϕ_{fa} definiert unter anderem auch, wie die von fa benutzten pspec-Terme, parm-Variablen, Sorten und Operationen auf die entsprechenden Teile in ai abgebildet werden. Darum werden alle von fa benutzten Terme und Variablen implizit über ϕ_{fa} mitaktualisiert. Wurde eine oder einer davon bereits in den Aktualisierungen$_{1...i-1}$ aktualisiert, so muß diese Aktualisierung mit ϕ_{fa} übereinstimmen.

Beispiel 5
Es wird zunächst eine Instanz von Arrayspec generiert, in der die Elemente elem durch Array-arrays aktualisert sind:

```
instantiate Arrayspec to Array=array-array
declare   (#el:Elemspec,
           #lm1:Limitspec(#in1:Indexspec),
           #lm2:Limitspec(#in2:Indexspec),
           #lm11:Limitspec(#in11:Indexspec),
           #lm21:Limitspec(#in21:Indexspec))

actualize #e by Array-array(#el,#lm1,#lm2,#lm11,#lm21)
    sorts elem by Array-array(#el,#lm1,#lm2,#lm11,#lm21).array
```

Es ergibt sich folgender Spezifikations-Kopf:

```
pspec Array=array-array (#el:Elemspec,
                         #liml:Limitspec(#indl:Indexspec),
                         #lim2:Limitspec(#ind2:Indexspec),
                         #lml:Limitspec(#inl:Indexspec),
                         #lm2:Limitspec(#in2:Indexspec),
                         #lmll:Limitspec(#inll:Indexspec),
                         #lm2l:Limitspec(#in2l:Indexspec))

   use pspecs   Array-array(#el,#lml,#lm2,#lmll,#lm2l)

   public sorts array
            ops new: → array
                put: #indl.index
                     #ind2.index
                     Array-array(#el,#lml,#lm2,#lmll,#lm2l).array
                     array   → array
                get: #indl.index #ind2.index array →
                     Array-array(#el,#lml,#lm2,#lmll,#lm2l).array
                isdef: #indl.index #ind2.index array → bool
                isnew: array → bool

   properties

   instantiates Arrayspec
                (#e:Elemspec,
                 #liml:Limitspec(#indl:Indexspec),
                 #lim2:Limitspec(#ind2:Indexspec))
   actualizes #e by Array-array(#el,#lml,#lm2,#lmll,#lm2l)
        sorts elem by array
```

Es soll nun eine Spezifikation Array-array=array-array generiert werden als Instanz von Array-array, in der die "Elemente" Arrayspec(#el,#liml1,#lim2l) aktualisiert werden sollen durch Arrayarrays. Dieses könnte erreicht werden, indem man wie oben den entsprechenden Signatur-Morphismus angibt. Doch genau dieser gewünschte Morphismus steht bereits in der instantiates-Klausel von Array=array-array beschrieben. Darum wird folgende Instanziierung unterstützt:

```
instantiate Array-array to Array-array=array-array
declare   (#ell:Elemspec,
           #lml:Limitspec(#inl:Indexspec),
           #lm2:Limitspec(#in2:Indexspec),
           #lmll:Limitspec(#inll:Indexspec),
           #lm2l:Limitspec(#in2l:Indexspec),
           #lmlll:Limitspec(#inlll:Indexspec),
           #lm2ll:Limitspec(#in2ll:Indexspec))

actualize Arrayspec(#el,#liml1,#lim2l) by instance
            Array=array-array(#ell,#lml,#lm2,
                              #lmll,#lm2l,#lmlll,#lm2ll)
```

Es gilt $\phi_{\text{Arrayspec(\#el,\#liml1,\#lim2l)}}$ zu bestimmen:

a. Bestimmung von $\tau_{\text{Arrayspec(\#el,\#lmll,\#lm2l)}}$

   #e → #el

   #liml → #liml1

   #indl → #indl1

   #lim2 → #lim2l

   #ind2 → #ind2l

   Arrayspec → Arrayspec(#el,#liml1,#lim2l)

b. Bestimmung von

$\tau_{\text{Array=array-array(\#e1l,\#lml,\#lm2,\#lmll,\#lm2l,\#lmlll,\#lm2ll)}}$:

```
#el → #ell
#liml → #lml
#indl → #inl
#lim2 → #lm2
#ind2 → #in2
#lml → #lmll
#inl → #inll
#lm2 → #lm2l
#in2 → #in2l
#lmll → #lmlll
#inll → #inlll
#lm2l → #lm2ll
#in2l → #in2ll
Array=array-array →
   Array=array-array(#ell,#lml,#lm2,#lmll,#lm2l,#lmlll,#lm2ll)
Array-array(#el,#lml,#lm2,#lmll,#lm2l) →
   Array-array(#ell,#lmll,#lm2l,#lmlll,#lm2ll)
```

c. Bestimmung von $\psi_{\text{Array=array-array}}$:

```
#e → Array-array(#el,#lml,#lm2,#lmll,#lm2l)
#liml → #liml
#indl → #indl
#lim2 → #lim2
#ind2 → #ind2
Arrayspec → Array=array-array
```

d. Bestimmung von $\phi_{\text{Arrayspec(\#el,\#lmll,\#lm2l)}}$:

```
#el → Array-array(#ell,#lmll,#lm2l,#lmlll,#lm2ll)
#lml → #lml
#inl → #inl
#lm2 → #lm2
#in2 → #in2
Arrayspec(#el,#lmll,#lm2l) →
```

Array=array-array(#ell,#1ml,#1m2,#1mll,#1m2l,#1mlll,#1m2ll)

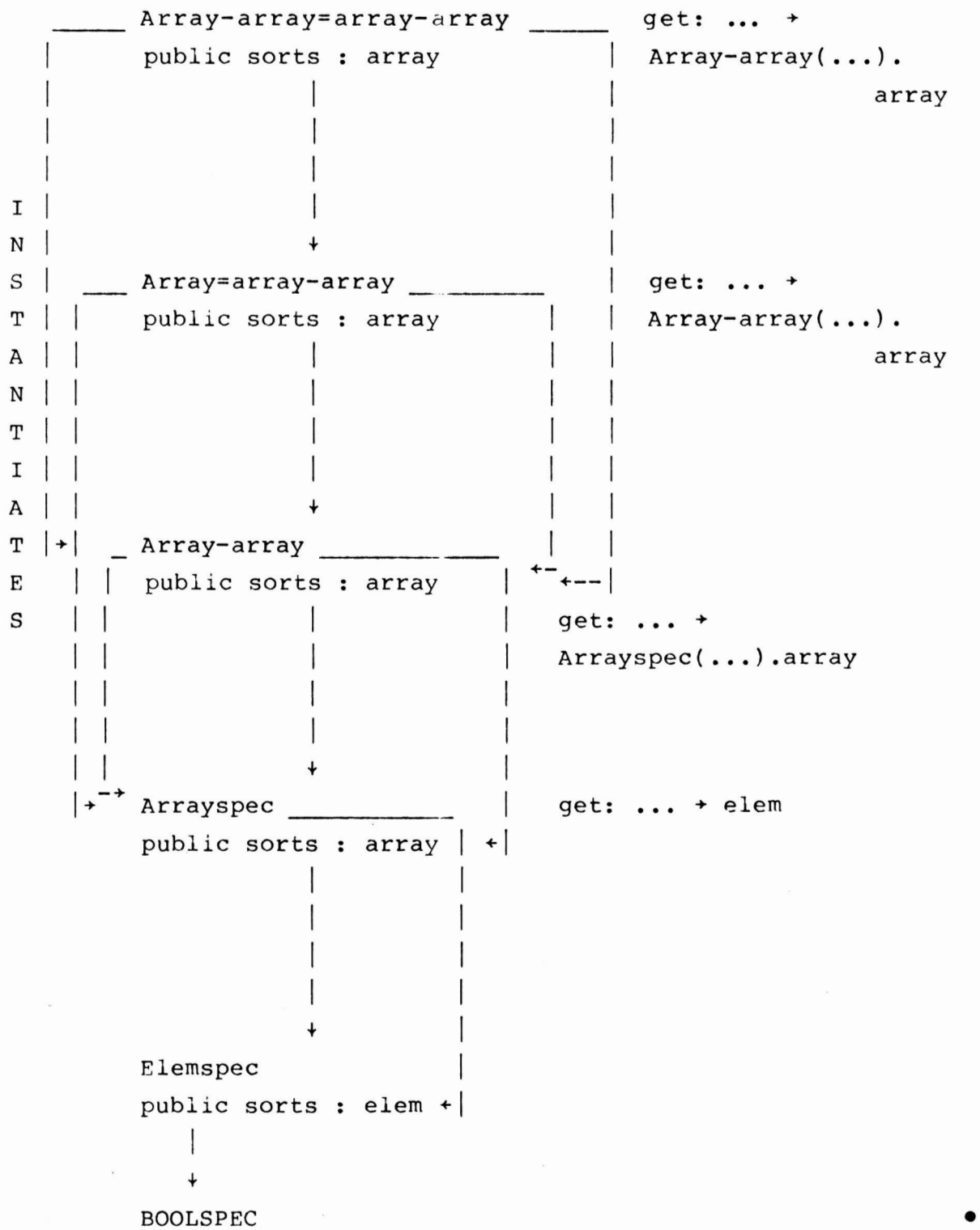
ϕArrayspec(#el,#1mll,#1m2l) existiert und deshalb ist die Aktualisierung korrekt. Implizit wird über ϕ ebenfalls die von Arrayspec(#el,#1mll,#1m2l) benutzte parm-Variable #el durch Array-array(#ell,#1mll,#1m2l,#1mlll,#1m2ll) aktualisiert. Die Spezifikation Array-array=array-arrray hat folgenden Spezifikationskopf:

```
pspec Array-array=array-array (#el:Elemspec,
                               #liml:Limitspec(#indl:Indexspec),
                               #lim2:Limitspec(#ind2:Indexspec),
                               #lml:Limitspec(#inl:Indexspec),
                               #lm2:Limitspec(#in2:Indexspec),
                               #lmll:Limitspec(#inll:Indexspec),
                               #lm2l:Limitspec(#in2l:Indexspec),
                               #lmlll:Limitspec(#inlll:Indexspec),
                               #lm2ll:Limitspec(#in2ll:Indexspec))

  use pspecs Array=Array-array(#ell,#1ml,#1m2,#1mll,#1m2l,#1mlll,#1m2l

   public sorts array
            ops new: → array
                put: #indl.index
                     #ind2.index
                     Array-array(#el,#1ml,#1m2,#1mll,#1m2l,#1mlll,
                     #1m2ll).array
                     array   → array
                get: #indl.index #ind2.index array →
                     Array-array(#el,#1ml,#1m2,#1mll,#1m2l,#1mlll,
                     #1m2ll).array
                isdef: #indl.index #ind2.index array → bool
                isnew: array → bool

   properties
```

```
instantiates Array-array
             (#el:Elemspec,
              #liml:Limitspec(#indl:Indexspec),
              #lim2:Limitspec(#ind2:Indexspec),
              #liml1:Limitspec(#indl1:Indexspec),
              #lim21:Limitspec(#ind21:Indexspec))
actualizes Arrayspec(#el,#liml1,#lim21) by instance
                Array=array-array(#el1,#lml,#lm2,
                                  #lml1,#lm21,#lml11,#lm211)
```

Anmerkung:
An der Operation get: #indl.index #ind2.index array → Array-array (#el,#lml,#lm2,#lml1,#lm21,#lml11,#lm211).array wird deutlich, daß die in der Aktualisierungs-Klausel dokumentierten Abbildungen nicht unbedingt gleichbedeutend sind mit der Ersetzung der Präfizes von Sorten und Operationen. Die Operation get: #indl.index #ind2.index array → Arrayspec(#el,#liml1,#lim21).array in Array-array geht hervor aus der Operation get: #indl.index #ind2.index array → elem in Arrayspec. Die Zielsorte elem von get in Arrayspec ist in Array-array=array-array wegen #e → #el → Array-array(#el1,#lml1,#lm21,#lml11,#lm211) abgebildet auf Array-array (#el1,#lml1,#lm21,#lml11,#lm211).array. Aus diesem Grund wird aus dem Präfix Arrayspec(#el,#liml1,#lim21) das Präfix Array-array (#el1,#lml1,#lm21,#lml11,#lm211), obwohl in der Aktualisierung die Abbildung Arrayspec(#el,#liml1,#lim21) → Array=array-array (#el1,#lml,#lm2,#lml1,#lm21,#lml11,#lm211) durchgeführt wurde.
Der hierarchische Aufbau von Array-array=array-array und die Zielsorte der Operation get in der jeweiligen Spezifikation werden an folgendem Diagramm veranschaulicht:

Zielsorte der Operation get in den jeweiligen Sezifikationen:



Definition

Die Instanziierungs-Abbildung Φ: specalt → specneu ist definiert für das downwards-interface von specalt durch die Vereinigung aller Abbildungen ϕ_{fa}, die durch die Aktualisierungen$_{1...n}$ festgelegt wurden. •

Definition

Jeder pspec-Spezifikationskörper definiert einen kanonischen Term-Funktor (vgl. [BG 80][BV 83]). Das CTF-requirement besagt: Die durch den formalen Parameter nach Anwendung von Φ definierten CTF-Funktoren werden auch durch den aktuellen Parameter definiert. •

Definition

Das property-requirement ist erfüllt für einen formalen Parameter fa, wenn nach Anwendung von ϕ_{fa} auf die properties von fa, diese impliziert werden durch die properties und Operations-Definitionen des aktuellen Parameters ai. •

Theorem

Die Instanziierungs-Abbildung Φ ist ein Spec-Morphismus, wenn das property-requirement und das CTF-requirement erfüllt sind. •

Definition

Die Instanziierung ist korrekt, wenn:

1. Φ ein Spec-Morphismus ist.
2. das Hierarchie-requirement erfüllt ist. •

Anmerkung:

1. Jede elementare Parameter-Spezifikation muß eine public-clause haben, da sie sonst als sspec geschrieben werden könnte. Ist nun specalt ein zusammengesetzter Parameter ohne public-clause, dann dürfen nicht alle formalen Parameter von specalt durch sspecs aktualisiert werden, da sonst specneu ein elementare Parameter ohne public-clause wäre.
2. Das Instanziierungs-Kommando garantiert, daß die Instanziie-

rungs-Abbildung Φ ein Signatur-Morphismus ist. Die Sicherstellung des property-requirements wird von Beweisern übernommen, die zur Zeit in SPESY integriert werden.

4. rename renaming

Wie die vorhergehenden Beispiele zeigen kann es nötig sein, Sorten- und Operationssymbole von specneu durch Voranstellen von Präfizes eindeutig zu machen, wobei unter Umständen recht lange Namen generiert werden. Um dieses zu verhindern, besteht die Möglichkeit, die Sorten- und Operationssymbole von specalt umzubenennen. Neben der Vermeidung von Präfizes können so für die instanziierten Symbole bedeutungsvolle Namen eingeführt werden. Das renaming hat folgende Form :

rename sorts as_1 by ns_1
.
.
as_n by ns_n
ops ao_1 by no_1
.
.
ao_m by no_m

$as_i \in \{as_1 ... as_n\}$ müssen Symbole aus den public-sorts, und $ao_i \in \{ao_1 ... ao_m\}$ müssen Symbole aus den public-operations von specalt sein.
$ns_i \in \{ns_1 ... ns_n\}$ müssen eindeutige Sortensymbole sein, die verschieden sind von allen nicht umbenannten Sortensymbolen von specalt.
$no_i \in \{no_1 ... no_m\}$ müssen eindeutige Operationssymbole sein, die verschieden sind von allen nicht umbenannten public- und private-operations von specalt.
$ns_i \in \{ns_1 ... ns_n\}$ und $no_i \in \{no_1 ... no_m\}$ sind die neuen Namen für die entsprechenden Sorten und Operationen in specneu.

Beispiel 6

Arrayspec soll wie in Beispiel 4 instanziiert werden zu einem Array, dessen Elemente wiederum aus Arrays bestehen. Die Sorte array der Instanz wird umbenannt in outerarray, um auf das Präfizieren der Sortensymbole verzichten und die Sorte, die das äußere Array bezeichnet von der, die das innere bezeichnet, besser unterscheiden zu können. Außerdem soll die erzeugende Operation generate statt new heißen.

```
instantiate Arrayspec to Array-array1
declare   (#el:Elemspec,
           #liml1:Limitspec(#indl1:Indexspec),
           #lim2l:Limitspec(#ind2l:Indexspec))
actualize #e by Arrayspec(#el,#liml1,#lim2l)
    sorts elem by array
rename sorts array by outerarray
       ops new   by generate
```

Das Ergebnis der Instanziierung ist folgende Spezifikation:

```
pspec Array-array1 (#el:Elemspec,
                    #liml:Limitspec(#indl:Indexspec),
                    #lim2:Limitspec(#ind2:Indexspec),
                    #liml1:Limitspec(#indl1:Indexspec),
                    #lim2l:Limitspec(#ind2l:Indexspec))

   use pspecs   Arrayspec(#el,#liml1,#lim2l)

   public sorts outerarray
          ops generate: → outerarray
              put:#indl.index#ind2.index array outerarray →
                    outerarray
              get: #indl.index #ind2.index outerarray →
                    array
```

```
            isdef: #ind1.index #ind2.index outerarray → bool
            isnew: outerarray → bool

properties

instantiates Arrayspec
            (#e:Elemspec,
             #lim1:Limitspec(#ind1:Indexspec),
             #lim2:Limitspec(#ind2:Indexspec))
actualizes #e by Arrayspec(#el,#lim11,#lim21)
     sorts elem by array
rename sorts array by outerarray
         ops new   by generate

constructors *generate, *put

auxiliaries  $bound1: outerarray → ind1.index
             $bound2: outerarray → ind2.index

define auxiliaries
   $bound1(a) :=
             case a is
                *generate : #ind1.min
                *put(m,n,e,al) : m
             esac

   $bound2(a) :=
             case a is
                *generate : #ind2.min
                *put(m,n,e,al) : #ind2.max(n,$bound2(al))
             esac
```

```
define carriers
   is-outerarray(a) :=
            case a is
               *generate : true
               *put(m,n,e,al) :
                  if not is-outerarray(al)
                  then false
                  else and(or(#indl.less($boundl(al),m),
                              and(eq.#indl.index(m,$boundl(al)),
                                  #ind2.less($bound2(al),n))),
                           and(#indl.less(m,#liml.limit),
                               #ind2.less(n,#lim2.limit)))
            esac

define constructors
   generate := *generate

   put(m,n,e,al) :=
            if and(or(#indl.less($boundl(al),m),
                      and(eq.#indl.index(m,$boundl(al)),
                          #ind2.less($bound2(al),n))),
                   and(#indl.less(m,#liml.limit),
                       #ind2.less(n,#lim2.limit)))
            then *put(m,n,e,al)
            elsif not(and(#indl.less(m,#liml.limit),
                          #ind2.less(n,#lim2.limit)))
            then error.outerarray
            else case al is
                    *generate : error.outerarray
                    *put(ml,nl,el,a2) :
                         put(ml,nl,el(put(m,n,e,a2)))
                 esac
```

```
define ops
   get(m,n,al) :=
            case al is
               *generate : error.array
               *put(ml,nl,el,a2) :
                if and(eq.#indl.index(m,ml),eq.#ind2.index(n,nl))
                then el
                elseif or(#indl.less(m,ml),
                             and(eq.#indl.index(ml,m),
                                 #ind2.less(n,nl))
                then get(m,n,a2)
                else error.array
            esac

   isdef(m,n,al) :=
            case al is
               *generate : error.bool
               *put(ml,nl,e,a2) :
                if and(eq.#indl.index(m,ml),eq.#ind2.index(n,nl))
                then true
                elseif or(#indl.less(m,ml),
                             and(eq.#indl.index(ml,m),
                                 #ind2.less(n,nl))
                then isdef(m,n,a2)
                else error.bool
            esac

   isnew(al) :=
            case al is
               *generate : true
               *put(m,n,e,a2) : false
            esac

endspec
```

Anmerkung:
Neben der Ersetzung aller umbenannten Sorten- und Operationssymbole in der Instanz, sind auch all die Symbole neu zu generieren, die aus diesen hervorgehen: *generate, is-outerarray, error.outarray.
Die hier aufgeführten Beispiele sind mit SPESY nachgerechnet worden. Das Protokoll von dieser SPESY-Sitzung befindet sich im Anhang B. •

Nach der fehlerfreien Abarbeitung wird dem instantiate-Kommando entsprechend eine neue Spezifikation specneu generiert:

1. Typ von specneu
   1. specalt war ein parm. Dann ist auch specneu ein parm.
   2. specalt war eine pspec. Specneu ist
      - eine sspec, wenn alle parm-Variablen von specalt aktualisiert wurden, und alle formalen Parameter durch sspecs aktualisiert wurden.
      - eine pspec im anderen Fall.

2. Parameter-Deklaration von specneu
   Die Parameter-Deklaration von specneu besteht aus der im instantiate-Kommando eingeführten actual-parm-declaration und der Deklaration aller nicht aktualisierten parm-Variablen aus der Parameter-Deklaration von specalt.

3. use-clause von specneu
   1. specneu benutzt folgende sspecs:
      - alle von specalt benutzten sspecs;
      - alle für die Aktualisierung direkter parm-Variablen von specalt benutzten sspecs;
      - alle sspecs, die für die Aktualisierung der von specalt benutzten pspec-Terme verwandt wurden.

2. specneu benutzt alle direkten parm-Variablen aus der Parameter-Deklaration von specneu, die nicht in von specneu benutzten pspec-Termen auftreten.
3. specneu benutzt folgende pspec-Terme:
   - alle Terme, die aktuelle Parameter sind für direkte parm-Variablen oder pspec-Terme von specalt;
   - alle nicht aktualisierten pspec-Terme von specalt; in diesen Termen werden die Variablen durch die reassigments bzgl. der Parameter-Deklaration von specneu ersetzt.
4. Wiederholtes Auftreten derselben sspec, parm-Variable oder desselben pspec-Terms wird gelöscht. Ebenso werden die Terme und sspecs gelöscht, die von anderen aus der use-Klausel benutzt werden.

4. public-clause von specneu
   Die public-clause von specneu geht hervor aus der von specalt durch Anwendung von Φ auf die opheaders. Mehrdeutig gewordene Sortennamen werden durch Präfizieren eindeutig gemacht. Im renaming umbenannte Sorten- und Operationssymbole werden ersetzt.

5. public-descriptive-clause von specneu
   1. Die properties von specneu gehen aus den properties von specalt hervor durch Anwendung von Φ auf diese. Operationssymbole, auf die ein renaming ausgeführt wurde, werden entsprechend umbenannt.
   2. Das instantiate-Kommando wird in die instantiates-Klausel konvertiert, welche die Instanziierung dokomentiert.

6. Der Spezifikationskörper von specneu geht aus dem von specalt hervor. Wurde ein Sortensymbol sa oder ein Operationssymbol oa aktualisiert oder im renaming umbenannt durch das Sortensymbol sn bzw. das Operationssymbol on, so sind folgende Ersetzungen für die Sortensymbole durchzuführen:

```
sa → sn in: auxiliaries
            private ops

$.eq.sa → $.eq.sn in: define auxiliaries
                      define carriers (falls sa public ist im
                                       entsprechenden formalen
                                       Parameter)

is-sa → is-sn in: define carriers
error.sa → error.sn in: define auxiliaries
                        define carriers
                        define constructors
                        define ops
```

Für die Operationsymbole:

1. Für alle oa, die nicht als *oa unter constructors aufgeführt sind:

1.1. Für alle oa, die nicht als $oa unter auxiliaries aufgeführt sind:

```
oa → on in: define auxiliaries
            define carriers
            define constructors
            define ops
```

1.2. Für alle oa, die als $oa unter auxiliariers aufgeführt sind:

```
$oa → $on in: auxiliaries
              define auxiliaries
              define carriers
oa → on in: define constructors
            define ops
```

2. Für alle oa, die als *oa unter constructors aufgeführt sind:

```
*oa → *on in: constructors
              define auxiliaries
              define carriers
              define constructors
              define ops
oa → on in: define carriers
            define constructors
            define ops
$is-*oa → $is-*on in: define auxiliaries
                      define carriers
is-*oa → is-*on in: define constructors
                    define ops.
```

3.3. OK-Bedingungen für die syntaktischen Teile einer ASPIK-Spezifikation

Für jeden Spezifikationstyp gibt es eine Reihe von Bedingungen, die erfüllt sein müssen, damit eine Spezifikation syntaktisch korrekt ist. Ein in das Spezifikationssystem SPESY integriertes CHECK-Modul testet eine Spezifikation auf die Erfüllung der folgenden Bedingungen und notiert das Ergebnis dieser Tests in einem ihr zugehörigen controlvector.

3.3.1. OK-Bedingungen für die syntaktischen Teile einer sspec SPEC

1. use-clause
   sspec-uses:
   - alle von SPEC benutzten Spezifikationen sind ok;
   - jede aufgeführte Spezifikation kommt genau einmal vor und bezeichnet eine sspec;
   - alle unter restrict to aufgeführten Operationsnamen bezeichnen public-operations der zugehörigen benutzten Spezifikation;
   - es treten keine Zyklen in der use-relation auf;
   - für die ausgezeichnete Spezifikation BOOLSPEC ist die use-clause leer; für jede andere Spezifikation ist die use-clause nicht leer, d.h. BOOLSPEC wird immer direkt oder indirekt benutzt.

2. public sorts
   - jeder aufgeführte Name kommt genau einmal vor und ist verschieden von allen Namen der verfügbaren Spezifikationen;
   - die Klausel kann leer sein.

3. public ops
   - jeder aufgeführte Name kommt genau einmal vor und ist verschieden von allen Symbolen aus public sorts und den Namen aller verfügbaren Spezifikationen;

- ist public sorts nicht leer, so darf auch public ops nicht leer sein, und es muß für jede Sorte s aus public sorts eine public operation existieren mit s als Zielsorte.
- alle Namen aus den Stelligkeiten und Zielsorten der Operationen bezeichnen Sorten aus dem upwards-interface von SPEC. Mehrdeutige importierte Namen müssen den Namen der sspec als Präfix erhalten, in deren public-clause sie definiert wurden. Das Symbol . trennt dabei Präfix und Sortennamen.

4. properties
   - die Klausel kann leer sein;
   - alle vorkommenden Symbole bezeichnen entweder Operationen aus dem upwards-interface von SPEC, oder Variablen. Mehrdeutige importierte Operationssymbole müssen den Namen der sspec als Präfix erhalten, in deren public-clause sie definiert wurden. Das Symbol . trennt dabei Präfix und Operationsnamen.

5. instantiation-clause
   - die Klausel wird automatisch generiert und ist deshalb immer ok.

6. constructors
   - alle vorkommenden Namen treten genau einmal auf;
   - für jeden auftretenden Namen gibt es eine entsprechende public operation, die eine Zielsorte aus public sorts hat;
   - ist public sorts leer, ist auch constructors leer;
   - für jede public Sorte s gibt es wenigstens einen constructor, dessen entsprechende public operation s als Zielsorte besitzt.

7. auxiliaries
   - die Klausel kann leer sein;
   - alle vorkommenden Namen treten genau einmal auf;
   - alle Namen aus den Stelligkeiten und Zielsorten der Operationen bezeichnen Sorten aus dem upwards-interface von SPEC.

Mehrdeutige Sortennamen müssen den Namen der sspec als Präfix erhalten, in deren public-clause sie definiert wurden. Das Symbol . trennt dabei Präfix und Sortennamen.
- gibt es eine Operation in der public-clause von SPEC, deren Name gleich dem einer auxiliary ist, so müssen Stelligkeit und Zielsorte beider Operationen übereinstimmen und es darf keinen constructor gleichen Namens geben;

8. define auxiliaries
   - ist auxiliaries leer, muß auch define auxiliaries leer sein;
   - die Funktionsdefinitionen enthalten keine constructors. Die Terme bestehen aus:
     - Variablen;
     - public operations aus dem downwards-interface von SPEC;
     - auxiliaries und vordefinierten auxiliaries; letztere werden automatisch generiert;
   - die in den Funktionsdefinitionen aufgeführten if-, let- und case-Schemata sind korrekt;
   - für die linke Seite jeder Funktionsdefinition gilt:
     - sie ist entweder eine 0-stellige auxiliary,
     - oder hat die Form: $a(x_1 \ldots x_n)$ und a ist eine auxiliary der Stelligkeit $s_1 \ldots s_n$ und x_i ist eine Variable der Sorte s_i; dabei müssen die Variablen verschieden voneinander sein.

9. define carriers
   - ist public sorts leer, muß auch define carriers leer sein;
   - für jede public sort s kann es eine charakteristische Funktion is-s:s--> bool geben;
   - die rechte Seite der Definition ist ein case-Schema;
   - die linke Seite der Definition hat die Form is-s(x), wobei x eine Variable der Sorte s ist. x ist die case-Variable des äußeren case-Schemas der rechten Seite;
   - die zur Definition verwendeten case-, if- und let-Schemata sind korrekt;

10. define constructors
   - ist public sorts leer, muß auch define constructors leer sein;
   - gibt es keine charakteristischen Funktionen, so hat jeder constructor $c:s_1...s_n-->s$ genau eine Definition der Form $c(x_1...x_n):=*c(x_1...x_n)$. Diese Funktionsdefinition wird automatisch erzeugt. Anderenfalls muß es genau eine Definition für jeden constructor c mit Zielsorte s geben, um c auf den carrier der Sorte s einzuschränken. Bei dieser Definition dürfen nur die public operations über den constructors, public operations aus dem downwards-interface und auxiliaries verwandt werden;
   - die benutzten case-, if- und let-Schemata sind korrekt.

11. private ops
   - die Klausel kann leer sein;
   - jeder in der Klausel aufgeführte Name kommt genau einmal vor und wurde noch nicht als Spezifikationsname, Sortenname im upwards-interface, oder als Operationsname in den public-operations oder als auxiliary verwandt;
   - alle Namen aus den Stelligkeiten und Zielsorten der Operationen bezeichnen Sorten aus dem upwards-interface von SPEC. Mehrdeutige Namen erhalten den Namen der sspec als Präfix, in deren public-clause sie definiert wurden. Das Symbol . trennt dabei Präfix und Sortennamen.
   - ist public ops leer, so muß auch private ops leer sein.

   Anmerkung: Alle auxiliaries, die nicht zu einer public operation korrespondieren, definieren implizit eine private operation.

12. define ops
   - jede public operation, die noch nicht unter define constructors oder implizit unter define auxiliaries definiert wurde, muß hier definiert werden;
   - jede Operation aus private ops muß hier definiert werden;

- für die linke Seite jeder Funktionsdefinition gilt:
  - sie ist entweder eine 0-stellige Operation,
  - oder hat die Form: $o(x_1 \ldots x_n)$ und o ist eine Operation der Stelligkeit $s_1 \ldots s_n$ und x_i ist eine Variable der Sorte s_i.
- die in den Funktionsdefinitionen vorkommenden Terme bestehen aus:
  - Variablen
  - operations aus dem upwards-interface
  - Operationen aus private ops
  - von auxiliaries abgeleitete private ops

  Die opids der letzten beiden Punkte müssen den Spezifikationsnamen als Präfix haben, wenn sie den gleichen Namen haben wie Operationen aus dem downward-interface.
- die verwendeten case-, if- und let-Schemata sind korrekt.

13. private-descriptive-clause
- die Klausel kann leer sein;
- die Terme bestehen aus:
  - Variablen
  - operations aus dem upward-interface
  - Operationen aus private ops
  - auxiliaries

14. document
- besteht aus 80 Zeichen langen Strings;
- die Klausel darf nicht leer sein.

15. case-scheme
- nach case steht eine Variable, die auf dem eindeutig existierendenPfad von der linken Seite der umgebenden Funktionsdefinitionen bis zu diesem case bereits vorgekommen ist, die aber nicht in einem let-Ausdruck eingeführt worden sein darf. Ihre Sorte muß eine public sort s sein.
- in den cases dürfen dann links nur constructors der Sorte s stehen. Sie enthalten entsprechend ihrer Definition die richtige Anzahl an Variablen als Argumente. Die Variablen

dürfen im Pfad noch nicht vorgekommen sein;
- auf der rechten Seite der cases dürfen nur op-schemes der Sorte s auftreten;
- gibt es ein otherwise, muß dort ein op-scheme der Sorte s stehen.

16. if-scheme
- auf if oder elseif muß ein Term der Sorte bool folgen;
- die auf then oder else folgenden op-schemes müssen von gleicher Sorte sein.

17. let-scheme
- für ein let-scheme der Form
  let xl=tl ... x_n=t_n in op-scheme gilt für i=1,...,n:
  - x_i sind Variablen;
  - t_i sind Terme;
  - die Sorte von x_i ist die Sorte von t_i
  - die Sorte des let-schemes ergibt sich aus der Sorte des op-schemes;
  - x_i darf auf dem eindeutig existierenden Pfad von der linken Seite der umgebenden Funktionsdefinition, die das let-scheme enthält, noch nicht vorkommen.

3.3.2. OK-Bedingungen für zusammengesetzte Teile einer sspec

1. public-clause
   (public sorts, public ops)
   - die Klausel ist ok, wenn public sorts und public ops ok sind;
   - die Klausel darf leer sein.

2. public-descriptive-clause
   (properties, instantiation-clause)
   - die Klausel ist ok, wenn die properties ok sind;
   - die Klausel darf leer sein.

3. spec-header
   (use-clause, public-clause, public-descriptive-clause)
   - der Teil darf nicht leer sein;
   - der Teil ist ok, wenn use-clause, public-clause und public-descriptive-clause ok sind.

4. auxiliaries-clause
   (auxiliaries, define auxiliaries)
   - die Klausel ist ok, wenn auxiliaries und define auxiliaries ok sind;
   - die Klausel darf leer sein.

5. carrier-part
   (constructors, auxiliaries-clause, define carriers)
   - die Klausel ist ok, wenn constructors, auxiliaries-clause und define carriers ok sind;
   - die Klausel muß leer sein, wenn public sorts leer ist.

6. ops-part
   (define constructors, private ops, define ops)
   - die Klausel ist ok, wenn define constructors, private ops und define ops ok sind;
   - die Klausel kann leer sein.

7. spec-body
   (carrier-part, ops-part, private-descriptive-clause)
   - der Teil ist ok, wenn carrier-part, ops-part und private-descriptive-clause ok sind;
   - der Teil darf leer sein.

8. gesamte Spezifikation
   - die Spezifikation ist ok, wenn spec-header, spec-body, und document ok sind.

3.3.3. OK-Bedingungen für die syntaktischen Teile einer pspec SPEC

1. parm-declaration
   - die Klausel darf nicht leer sein;
   - die Klausel wird von (und) eingeschlossen;
   - jede parm-variable beginnt mit #;
   - das erste Auftreten einer Variablen muß definierend sein, jedes weitere Auftreten muß davon abgeleitet sein;
   - alle über die parm-declaration benutzten parms müssen ok sein;
   - alle parm-variablen sind wohl definiert.

2. use-clause
   1) sspec-uses:
      - alle von SPEC benutzten Spezifikationen sind ok;
      - jede aufgeführte Spezifikation kommt genau einmal vor und bezeichnet eine sspec;
      - alle unter restrict to aufgeführten Operationsnamen bezeichnen public-operations der zugehörigen benutzten Spezifikation;
      - es treten keine Zyklen in der use-relation auf;
      - die Klausel kann leer sein.
   2) pspec-uses:
      - es treten nur Terme der Form spec($\#x_1,\ldots,\#x_n$) auf, wobei $\#x_i (i=1,\ldots,n)$ eine parm-variable aus der parm-declaration ist, und spec eine pspec bezeichnet, die ihrerseits ok ist;
      - jeder aufgeführte Term kommt genau einmal vor;
      - jeder aufgeführte Term ist wohl definiert;
      - alle unter restrict to aufgeführten Operationsnamen bezeichnen public-operations der zugehörigen benutzten Spezifikation;
      - es treten keine Zyklen auf;
      - die Klausel darf leer sein.

3) parm-uses:
   - die Klausel darf nicht leer sein;
   - die aufgeführten Symbole bezeichnen die direkten parm-variablen aus der parm-declaration;
   - die aufgeführten Variablen sind wohl definiert;
   - die benutzten parms sind ok.

3. Für alle anderen syntaktischen Teile einer pspec gelten die gleichen Bedingungen, wie für sspecs mit folgender Erweiterung: Mehrdeutige Sorten- und Operationssymbole aus den public-Klauseln von benutzten parms und pspecs müssen die entsprechenden parm-Variablen oder pspec-Terme als Präfix erhalten.

3.3.4. OK-Bedingungen für zusammengesetzte Teile einer pspec

1. public-clause
   (public sorts, public ops)
   - die Klausel ist ok, wenn public sorts und public ops ok sind;
   - die Klausel darf leer sein.

2. public-descriptive-clause
   (properties, instantiation-clause)
   - die Klausel ist ok, wenn die properties ok sind;
   - die Klausel darf leer sein.

3. spec-header
   (use-clause, public-clause, public-descriptive-clause)
   - der Teil darf nicht leer sein;
   - der Teil ist ok, wenn use-clause, public-clause und public-descriptive-clause ok sind.

4. auxiliaries-clause
   (auxiliaries, define auxiliaries)
   - die Klausel ist ok, wenn auxiliaries und define auxiliaries ok sind;
   - die Klausel darf leer sein.

5. carrier-part
   (constructors, auxiliaries-clause, define carriers)
   - die Klausel ist ok, wenn constructors, auxiliaries-clause und define carriers ok sind;
   - die Klausel muß leer sein, wenn public sorts leer ist.

6. ops-part
   (define constructors, private ops, define ops)
   - die Klausel ist ok, wenn define constructors, private ops und define ops ok sind;
   - die Klausel kann leer sein.

7. spec-body
   (carrier-part, ops-part, private-descriptive-clause)
   - der Teil ist ok, wenn carrier-part, ops-part und private-descriptive-clause ok sind;
   - der Teil darf leer sein.

8. gesamte Spezifikation
   - die Specifikation ist ok, wenn parm-declaration, spec-header, spec-body und document ok sind.

3.3.5. OK-Bedingungen für die syntaktischen Teile eines parms SPEC

1. parm-declaration
   - die Klausel darf leer sein;
   - die Klausel wird von (und) eingeschlossen;
   - jede parm-variable beginnt mit #;
   - das erste Auftreten einer Variablen muß definierend sein, jedes weitere Auftreten muß davon abgeleitet sein;
   - alle über die parm-declaration benutzten parms müssen ok sein;
   - alle parm-variablen sind wohl definiert.

2. use-clause
   1) sspec-uses:
      - die Klausel darf leer sein;
      - alle von SPEC benutzten Spezifikationen sind ok;
      - jede aufgeführte Spezifikation kommt genau einmal vor und bezeichnet eine sspec;
      - alle unter restrict to aufgeführten Operationsnamen bezeichnen public-operations aus dem upwards-interface der benutzten Spezifikationen;
      - es treten keine Zyklen in der use-relation auf;
   2) parm-uses:
      - die Klausel ist leer, wenn SPEC ein elementarer parm ist. Sonst sind hier alle direkten parm-variablen aus der parm-declaration aufgeführt;
      - alle benutzten parms sind ok.
   - beide zusammen dürfen nicht leer sein;

3. public sorts
   - jeder aufgeführte Name kommt genau einmal vor und ist verschieden von allen Namen der verfügbaren Spezifikationen;
   - die Klausel kann leer sein.

4. public ops
   - jeder aufgeführte Name kommt genau einmal vor und ist verschieden von allen Symbolen aus public sorts und den Namen aller verfügbaren Spezifikationen;
   - alle Namen aus den Stelligkeiten und Zielsorten der Operationen bezeichnen Sorten aus dem upwards-interface von SPEC. Mehrdeutige importierte Namen müssen den Namen der sspec als Präfix erhalten, in deren public-clause sie definiert wurden. Das Symbol . trennt dabei Präfix und Sortennamen.

5. Für properties und instantiation-clause gelten die gleichen Bedingungen, die für sspecs angegeben wurden.

3.3.6. OK-Bedingungen für zusammengesetzte Teile eines parms

1. public-clause
   (public sorts, public ops)
   - die Klausel ist ok, wenn public sorts und public ops ok sind;
   - die Klausel darf nur dann leer sein, wenn die Spezifikation kein elementarer parm ist.

2. public-descriptive-clause
   (properties, instantiation-clause)
   - die Klausel ist ok, wenn die properties ok sind;
   - die Klausel darf leer sein.

3. spec-header
   (use-clause, public-clause, public-descriptive-clause)
   - der Teil darf nicht leer sein;
   - der Teil ist ok, wenn use-clause, public-clause und public-descriptive-clause ok sind.

4. gesamte Spezifikation
   - die gesamte Spezifikation ist ok, wenn parm-declaration, spec-header und document ok sind.

3.4. Syntax von ASPIK

```
sspec ::=
        sspec specid
              sspec-header
              [spec-body]
              description
        endspec

sspec-header ::=
        sspec-use-clause
        [public-clause]
        [public-descriptive-clause]

sspec-use-clause ::=
        use sspec-uses

sspec-uses ::=
        sspecs (specid [restrict to opid* ])+

public-clause ::=
        public [sorts simple-id+ ]
               [ops op-header+ ]

op-header ::=
        opidl+: sortid* --> sortid

public-descriptive-clause ::=
        [properties property+ ]
        [instantiation-clause]

spec-body ::=
        [carrier-part]
         ops-part
        [private-descriptive-clause]
```

```
carrier-part ::=
        constructor-clause
        [auxiliaries-clause]
        [carrier-definition-clause]

constructor-clause ::=
        constructors  constrid+

auxiliaries-clause ::=
        auxiliaries aux-header+
        define auxiliaries aux-body+

aux-header ::=
        auxid+: sortid* --> sortid

aux-body ::=
        auxid[((varid,)*varid)]:= op-scheme |
        varid _n-opid_ varid := op-scheme

op-scheme ::=
        term | if-scheme | case-scheme | let-scheme

n-char ::=
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

if-scheme ::=
        if term
        then op-scheme
        (elseif term
         then op-scheme)*
        else op-scheme
```

```
case-scheme ::=
        case varid is
                (constrid[((varid,)* varid)]: op-scheme)+
                [otherwise op-scheme]
        esac

let-scheme ::=
        let (varid=term,)* varid=term in op-scheme

carrier-definition-clause ::=
        define carriers characteristic-function-definition+

characteristic-function-definition ::=
        characteristic-opid (varid) := case-scheme

ops-part ::=
        [define constructors constr-body+]
        [private ops op-header+]
        [define ops op-body+]

constr-body ::=
        constrid[((varid,)* varid)]:= op-scheme
        varid _n-constrid_ varid := op-scheme

op-body ::=
        opid[((varid,)* varid)]:= op-scheme |
        opid[[(varid,)* varid]]:= op-scheme |
        opid[<(varid,)* varid>]:= op-scheme |
        opid[{(varid,)* varid}]:= op-scheme |
        varid _n-opid_ varid := op-scheme

specid ::=
        simple-id | simple-id ((specid,)* specid)
```

simple-id ::=
a-char | a-char an-char*

a-char ::=
A | B | ...|Z

an-char ::=
a-char | n-char

sortid ::=
simple-id | specid.simple-id

constrid ::=
*n-opid | _*n-opid_

auxid ::=
$n-opid | _$n-opid_ | standard-auxid

opidl ::=
n-opid | _n-opid_ | standard-auxid

char ::=
an-char | | | [|] | < | > | { | } | (|) | * | § | % |
, | . | ! | @ | = | ^ | ; | | | : | _ | ´ | ` | ? | / | # |
- | + | " | &

n-opid ::=
char | a-char(an-char)* | {} | () | <> | [] |

```
property ::=
        equation

equation ::=
        |(term = term &)* term=term ==>| term eq-sign term

eq-sign ::=
        = | =/=

instantiation-clause ::=
        instantiates specid
        [actualizes actualization-clause+|
        [renames [sorts (simple-id by simple-id)+|
                 [ ops (opidl by opidl)+|

actualization-clause ::=
        formal+ by_match actual |
                --------

        formal+ by_instance actual |
                -----------

        formal+ by actual
               [sorts (simple-id by sortid)+|
               [ops (opidl by opid)+|

formal ::=
        p-varid | pspec-term

actual ::=
        formal | specid

opid ::=
        opidl | specid.opidl

characteristic-opid ::=
        is-simple-id
```

description ::=
("char^{80}")$^{+}$

standard-auxid ::=
$arg-(n-char)$^{+}$-constr-id | $standard-opid |
$is-constr-id

standard-opid ::=
eq. simple-id | error. simple-id

term ::=
varid | opid | auxid | constrid |
[n-opid | $n-opid | *n-opid | standard-opid | standard-auxid | specid.n-opid | specid.standard-opid](($(term,)$^{*} term) | term (n-opid | $n-opid | *n-opid | specid.n-opid] term | ((term,)* term) | |(term,)* term| | {(term,)* term} | <(term,)* term> | [(term,)* term| | *((term,)* term*) | *|(term,)* term*| | *{(term,)* term*} | *<(term,)* term*> | *[(term,)* term*|

parm ::=
parm specid[(parm-declaration)]
parm-header
endspec

parm-declaration ::=
((p-varid,)* p-varid:parm-term,)* (p-varid,)* p-varid:parm-term

p-varid ::=
#varid

parm-term ::=
specid[(component-parm-declaration)]

```
component-parm-declaration ::=
        ((p-varid,)* p-varid[:parm-term],)* (p-varid,)* p-varid
        [:parm-term]

parm-header ::=
        parm-use-clause
        [public-clause]
        [public-descriptive-clause]

parm-use-clause ::=
        use (sspec-uses|parm-uses)
        use sspec-uses
            parm-uses

parm-uses ::=
        parms p-varid+

pspec ::=
        pspec specid parm-declaration
              pspec-header
              [spec-body]
        endspec

pspec-header ::=
        pspec-use-clause
        [public-clause]
        [public-descriptive-clause]

pspec-use-clause ::=
        use [sspec-uses](parm-uses|pspec-uses) |
        use [sspec-uses]
            parm-uses
            pspec-uses

pspec-uses ::=
        pspecs (pspec-term[restrict to opid+])+
```

›spec-term ::=
specid (<u>(</u>p-varid<u>,</u>)* p-varid<u>)</u>

4.1. Benutzerhandbuch

4.1.1. Zweck und Anwendung des Systems

SPESY ist ein Software-Werkzeug zum unterstützten Erstellen, Editieren und Instanziieren, Speichern und Verwalten von in ASPIK geschriebenen Spezifikations-Hierarchien. Die bei der Systemkonzeption verwandte Modultechnik bietet die Anschlußmöglichkeit für eine Vielzahl weiterer Softwarewerkzeuge wie symbolische Interpretierer, Termersetzungssysteme, mechanische Beweiser, Programmtransformationssysteme usw. SPESY stellt somit ein ausbaufähiges Basissystem für einen umfassenden, integrierten Software-Entwicklungs-Arbeitsplatz dar.

Zur Basis von SPESY gehören folgende Komponenten:

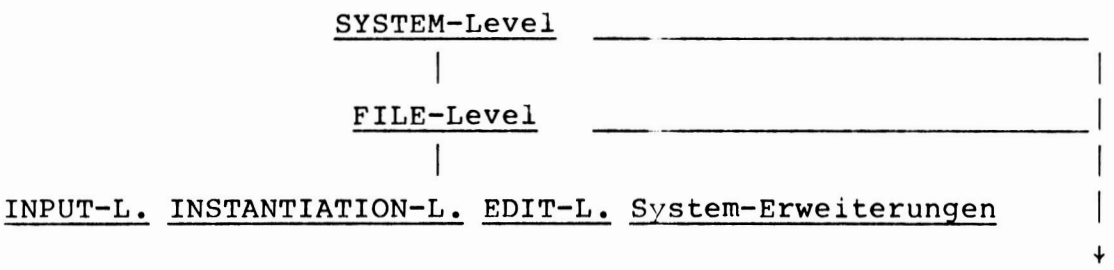
1. Datei- und Systemverwaltung mit Datenschutz;
2. Zwei Modi zur Eingabe von Spezifikationen:
   - voll unterstützt
   - teilweise unterstützt
3. Editor für Spezifikationen;
4. Voll unterstütztes Instanziieren von parametrischen Spezifikationen und Parameter-Spezifikationen;
5. Checker bzgl. der syntaktischen Korrektheit von Spezifikationen;
6. List- und Druckfunktionen für:
   - Spezifikationen
   - Teile von Spezifikationen
   - Informationen über Spezifikationen
7. Bereitstellung von Bibliotheken mit vordefinierten, korrekten Spezifikationen;

SPESY wurde auf einer SIEMENS 7.760 im BS2000 (Version 7.0) implementiert. Die Zielsprache ist SIEMENS-INTERLISP (Version 5.0).

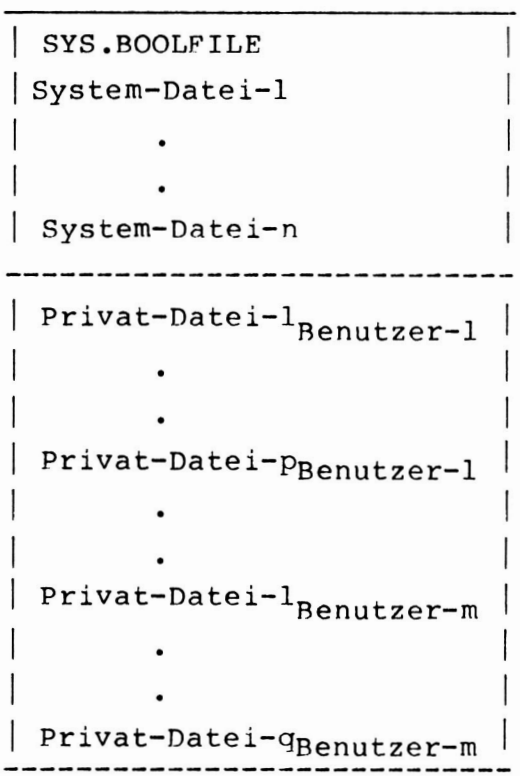
Anmerkung:
Eingabe, Editor sowie Teile des Checkers sind nicht Gegenstand dieser Arbeit. Detaillierte Information ist zu finden in [KRST 83 -1].

4.1.2. Allgemeine Systembeschreibung

Der Aufbau des Systems ist hierarchisch angeordnet:



KOMMANDO-EBENE



DATENBASIS

Neben den registrierten privaten Systembenutzern gibt es einen privilegierten Benutzer, dem als System-Manager die Sytemverwaltung obliegt, und der außerdem Eigentümer aller System-Dateien ist. Jeder Benutzer weist sich dem System gegenüber durch Angabe seiner Benutzerkennung und seines passwords aus. Die System-Dateien enthalten syntaktisch korrekte Spezifikationen, deren Namen eindeutig sind bzgl. aller anderen System-Dateien. Jeder Benutzer hat das Recht, auf diese Dateien lesend zuzugreifen und über den dort vorhandenen Spezifikationen neue Hierarchien zu definieren, die dann in einer dem Benutzer gehörenden Privat-Datei abgelegt werden.

Die Aufgaben des System-Managers sind:
1. Verwaltung der System-Dateien;
2. Verwaltung der Benutzer-Listen, in der alle Personen, die das Recht auf Systembenutzung haben, registriert sind;
3. Verwaltung der entsprechenden password-Listen.

Die Aufgaben des System-Managers werden auf dem SYSTEM-Level realisiert; darum sind die anderen Levels für ihn nicht verfügbar.

Es gibt eine ausgezeichnete System-Datei SYS.BOOLFILE, die nur die ausgezeichnete Spezifikation BOOLSPEC enthält. Jede andere Datei hat eine Dateiumgebung, das <u>environment</u>, bestehend aus einer nicht leeren Menge von Systemdateien. Alle Spezifikationen aus der Vereinigung der transitiven Hüllen der im environment aufgeführten System-Dateien stehen in der entsprechenden Datei zum Aufbau von Spezifikations-Hierarchien zur Verfügung. Auf diese Weise wird garantiert, daß die Spezifikation BOOLSPEC immer für jede andere Spezifikation verfügbar ist. Das environment wird für jede Datei bei deren Generierung festgelegt.

SYSTEM-Level

Hier findet hauptsächlich die System- und Dateiverwaltung statt:

1. Systemverwaltung
   Die Systemverwaltung erfolgt durch den System-Manager.Er kann Privat-Dateien der Benutzer konvertieren in System-Dateien, soweit die notwendigen Überprüfungen für die auf der Privat-Datei vorhandenen Spezifikationen auf Korrektheit hin und Eindeutigkeit bzgl. aller vorhandenen Spezifikationen auf den System-Dateien, keine Fehler erkennen lassen. Dem früheren Benutzer wird dabei das Eigentumsrecht über die Datei entzogen, und er erhält, wie alle anderen Benutzer auch, das Leserecht auf die Datei. Der System-Manager ist nun neuer Eigentümer.
   Der System-Manager kann außerdem System-Dateien löschen, soweit sie nicht im environment einer anderen Datei enthalten sind.
   Er kann neuen Benutzern das System verfügbar machen, indem er für diese einen Eintrag in die Benutzer- und password-Tabelle des Systems macht. Die neuen Benutzer erhalten das Leserecht auf alle System-Dateien.
   Er kann Benutzern das Recht auf die Systembenutzung entziehen. Alle Dateien sowie alle vorhandenen Rechte auf System- oder Privat-Dateien dieser Benutzer werden gelöscht.

2. Dateiverwaltung und Datenschutz
   Jeder Benutzer hat die Möglichkeit, beliebig viele Privat-Dateien, deren Eigentümer er ist, anzulegen und diese auch wieder zu löschen. Die Dateien sind entweder leer oder enthalten ASPIK-Spezifikationen. Um Namenskonflikte zu vermeiden, trägt jede Datei die Benutzerkennung des Besitzers durch einen Punkt getrennt als Präfix. Bezieht sich ein Kommando auf eine Datei des Besitzers, so muß dieser das Präfix nicht zum Dateinamen angeben. Bezieht sich ein Kommando auf eine fremde Datei, ist der vollqualifizierte Dateiname -also mit Präfix- anzugeben.
   Dateien sind grundsätzlich vor unberechtigtem Zugriff durch

fremde Benutzer geschützt. Daneben ist es dem Benutzer möglich, explizit Lese- oder Schreibrechte auf eine Datei an alle bzw. einzelne fremde Benutzer zu vergeben, oder bereits vorhandene Rechte an einer Datei zu entziehen oder zu ändern. Er selbst und der System-Manager haben immer Lese- und Schreibrecht bzgl. der Datei. Diese können weder geändert noch entzogen werden.
Das Leserecht an einer fremden Datei bezieht sich stets auf die Datei als Ganzes. Das Leserecht auf einer Datei gibt einem Benutzer die Möglichkeit, diese zu kopieren. Um Spezifikationen aus einer fremden Datei verändern oder benutzen zu können, muß diese zuvor immer kopiert werden. So ist garantiert, daß jedem Benutzer sämtliche Spezifikationen von allen anderen Benutzern verfügbar gemacht werden können, auf der anderen Seite aber niemand einem fremden Benutzer etwas zerstören kann.
Ein dem System angeschlossenes message-System [KRST 83-1][KRST 83-2] erfordert die Unterscheidung zwischen Lese- und Schreibrecht. Hat ein Benutzer das Schreibrecht auf eine fremde Datei, kann er dort Nachrichten für den Eigentümer ablegen.

Der Benutzer kann sich durch SPESY listings von Spezifikationen erzeugen lassen oder aber auch eine Sitzung ganz oder teilweise protokollieren lassen. Der output wird in entsprechenden Dateien bis zum Ausdruck zwischengespeichert. Derartige output-Dateien sind durch SPESY-Kommandos weder verfügbar noch manipulierbar.

Dem OPEN-Kommando kommt eine mehrfache Bedeutung zu:
- Der SYSTEM-Level wird verlassen und der FILE-Level betreten, d.h. nach Ausführung dieses Kommandos sind die Kommandos des SYSTEM-Levels nicht mehr, dafür aber die Kommandos des FILE-Levels verfügbar. Die Objekte, auf denen die Kommandos arbeiten, sind nun nicht mehr die Systemobjekte wie Dateien, sondern die Objekte einer Datei, nämlich die Spezifikationen.
- Bezeichnet der zum OPEN-Kommando gehörende Parameter file eine noch nicht im Dateikatalog des Benutzers vorhandene Datei, so wird eine solche generiert, und das environment für die Datei

wird festgelegt. Alle Spezifikationen aus dem environment stehen dem Benutzer zur Verfügung, um in anderen Spezifikationen benutzt zu werden. Die Spezifikationen aus dem environment dürfen aber nicht editiert werden.
- Bezeichnet file eine bereits vorhandene Datei, so stehen dem Benutzer neben den Spezifikationen des environments auch die von file zur Verfügung.
- System-Dateien dürfen nicht geöffnet werden; damit kann der System-Manager den SYSTEM-Level nicht verlassen und Systemspezifikationen sind vor Veränderung durch den Editor geschützt.

FILE-Level

Alle Kommandos des FILE-Levels beziehen sich auf die verfügbaren Spezifikationen, das sind die Spezifikationen aus dem environment und der geöffneten Privat-Datei.

- verschiedene LIST-Kommandos geben Auskunft über Spezifikationen und Beziehungen zwischen Spezifikationen;
- mit dem CHECK-Kommando kann eine Spezifikation auf syntaktische Korrektheit hin überprüft werden;
- das PRINT-Kommando schreibt Spezifikationen auf eine output-Datei und veranlaßt deren Ausdruck;
- mit dem INPUT-Kommando wird der FILE-Level verlassen und der INPUT-Level betreten; die Kommandos des FILE-Levels sind nicht mehr verfügbar. Der Benutzer kann eine neue Spezifikation voll- oder teilunterstützt eingeben. Eine Spezifikation mit diesem Namen darf sich noch nicht unter den verfügbaren Spezifikationen befinden. Die neuerstellte Spezifikation wird nach Abarbeitung des Kommandos in der geöffneten Privat-Datei des Benutzers aufgenommen. Nach Abarbeitung des Kommandos befindet sich der Benutzer wieder auf dem FILE-Level.

- mit dem INSTANTIATE-Kommando wird der FILE-Level verlassen und der INSTANTIATION-Level betreten. Die Kommandos des FILE-Levels sind nicht mehr verfügbar. Der Benutzer hat die Möglichkeit eine verfügbare Spezifikations-Hierarchie mit einer pspec oder einem parm an der Spitze systemgesteuert ganz oder partiell zu instanziieren. Endet die Instanziierung fehlerfrei, so wird die Instanz in der geöffneten Privat-Datei des Benutzers aufgenommen. Nach Abarbeitung des Kommandos befindet sich der Benutzer wieder auf dem FILE-Level.
- mit dem EDIT-Kommando wird der FILE-Level verlassen und der EDIT-Level betreten. Die Kommandos des FILE-Levels sind nicht mehr verfügbar. Es kann eine Spezifikation aus der geöffneten Privat-Datei des Benutzers editiert werden. Sollen die Änderungen erhalten bleiben, so wird die ursprüngliche Spezifikations-Definition überschrieben; im anderen Fall hat das EDIT-Kommando keine Seiteneffekte. Nach Abarbeitung des Kommandos befindet sich der Benutzer wieder auf dem FILE-Level.

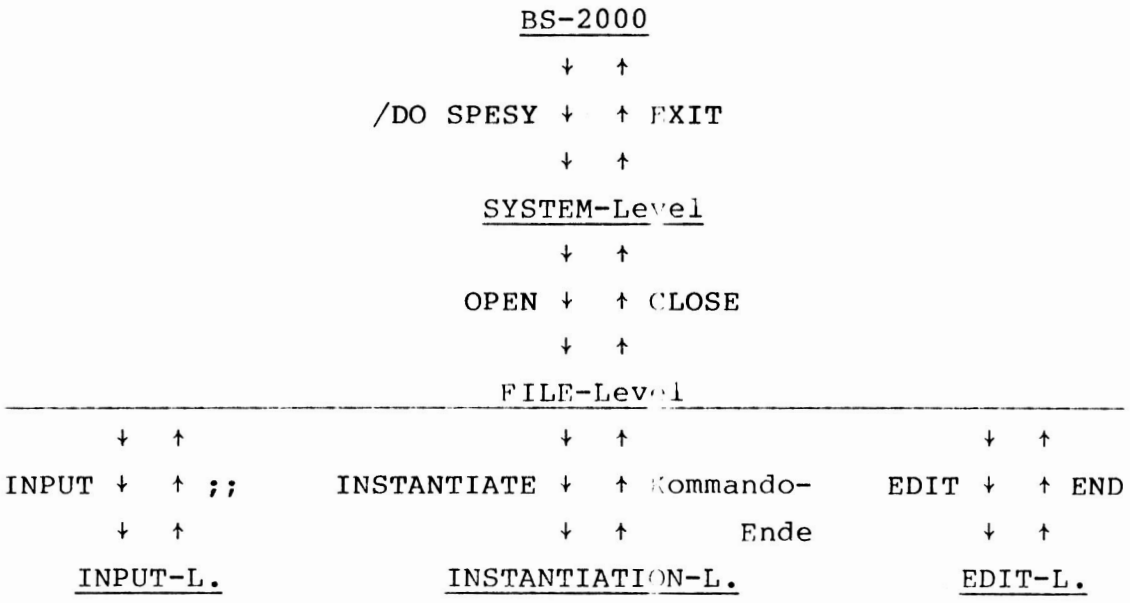
Auf jedem Level gibt es ein HELP-Kommando, das Informationen über die verfügbaren Kommandos liefert. Verlangt ein Kommando Parameter, so können diese direkt vom Benutzer angegeben werden. Fehlen die Parameterangaben, werden sie vom System angefordert.

Die Promptsymbole für die einzelnen Level sind:

SYSTEM-Level	: Ready:
FILE-Level	: ENTER CMD:
INPUT-Level	: verschiedene spezielle Anforderungen
INSTANTIATION-Level	: verschiedene spezielle Anforderungen
EDIT-Level	: CMD:

Der Systemaufruf erfolgt im BS-2000 durch /DO SPESY, wohin man auch nach Verlassen des Systems durch das SYSTEM-Level-Kommando EXIT zurückgelangt.

Der Übergang zwischen den Leveln stellt sich graphisch folgendermaßen dar:



4.1.3. Hinweise zum Betrieb von SPESY

4.1.3.1. Systemaufruf

Der Systemaufruf erfolgt auf der BS-2000-Betriebssystemebene durch:

```
/DO SPESY
```

Dieser Aufruf bewirkt, daß zunächst die Funktionstasten am Terminal-Keyboard folgende Belegung erhalten:

1 OPEN	2 EDIT	3 DEFINE
4 CLOSE	5 END	6 ENDSPEC
7 LISTR	8 ENTRYR	9 REMOVER
10 LISTC	11 LISTALLSPECS	12 LIST
13 NAMEORIGINE	14 ANYCYCLES?	15 STOP
16 LSPUSING	17 LSPUSEDBY	18 EXIT
19 ;;	20 -->	P

Anschließend werden SIEMENS-INTERLISP und SPESY geladen. Der Benutzer wird nach seiner Benutzerkennung und seinem password gefragt und erreicht anschließend den SYSTEM-Level. Sollte zur gleichen Zeit ein anderer Benutzer mit SPESY arbeiten, wird die Warnung:

```
****************************************
* SPESY IN USE - DON´T CHANGE ANYTHING *
****************************************
```

ausgegeben. In diesem Fall darf der Benutzer keine Kommandos ausführen, die eine Änderung der Systemlisten (SYSTEMLOG) zur Folge hätten. Die verbotenen Kommandos sind:

SYSTEM-Level : COPYF
DELETEF
ENTRYR
Anlegen neuer Dateien mit OPEN
REMOVER

FILE-Level : COPYS
DELETES
Ändern von Spezifikationen mit EDIT
INPUT
INSTANTIATE

Ist der SYSTEM-Level erreicht, meldet SPESY sich mit:

SPECSYSTEM (VERS.1.0) GENERATED ON <date><time> IS READY:

und erwartet die Eingabe von SYSTEM-Level-Kommandos.

4.1.3.2. Verhalten bei Fehlern

Sollte einmal ein Fehler auftreten, so übernimmt das INTERLISP-System die weitere Steuerung. Auf dem Bildschirm erscheint eine Fehlermeldung und das prompt-Symbol:

```
1:
```

Hier muß der Benutzer folgendes Kommando eingeben:

```
(SPESY.ERROR)
```

Für den Fall, daß der Benutzer beim Auftreten des Fehlers ein Protokoll führt, wird dieses beendet. In diesem Fall, oder wenn während der Sitzung protokolliert wurde, fragt das System, ob ein Ausdruck des Protokolls vom Benutzer gewünscht wird. Ist das der Fall, wird der Ausdruck veranlaßt und die Protokoll-Output-Datei nach dem Ausdruck gelöscht. Wünscht der Benutzer keinen Ausdruck, wird die Größe der Datei auf ein Minimum reduziert und steht dem Benutzer unter dem Dateinamen Benutzername.PROTOCOL zur Verfügung. SPESY kann auf diese Datei nicht mehr zugreifen. Der Benutzer befindet sich anschließend auf der BS2000-Betriebssystemebene.

Sollte es beim Aufruf von /DO SPESY zu Fehlern kommen, kann der Grund dafür nur in einer nicht vollständigen Abarbeitung des SPESY-Bootstraps beim davor liegenden SPESY-Aufruf, z.B. durch einen Maschinenfehler, zu finden sein. In diesem Fall erfolgt eine System-Initialisierung durch

```
/DO SPESY.FAIL
```

Ein anschließender Aufruf von /DO SPESY führt dann zum gewünschten Ergebnis.

4.1.4. Die Kommandos von SPESY

4.1.4.1. SYSTEM-Level-Kommandos

Es wird unterschieden zwischen privilegierten Kommandos, die nur demSystem-Manager und Kommandos, die allen Benutzern zur Verfügung stehen. Die Kommandos werden auf Ausführbarkeit hin untersucht und entweder ausgeführt, oder es erfolgt im Fehlerfall eine entsprechende Fehlermeldung.

4.1.4.1.1. Privilegierte Kommandos

Die privilegierten Kommandos sind:

ADDUSPW
CHANGEPW
LISTUSPW
REMUSID

Kommandoname : ADDUSPW alias: AU
Parameter : <userid> <psword>

Das Kommando generiert einen Eintrag für einen Benutzer unter der Benutzerkennung userid in der Liste der zugelassenen Benutzer. psword wird als password für userid in der password-Tabelle von SPESY aufgenommen. userid darf noch nicht in der Benutzertabelle vermerkt sein. userid und psword dürfen bis zu 8 Zeichen lang sein.

Kommandoname : CHANGEPW alias: CP
Parameter : <userid> <psword>

Das Kommando ordnet dem bereits vorhandenen Benutzer userid den Neueintrag psword in der password-Tabelle von SPESY zu. psword darf bis zu 8 Zeichen lang sein.

Kommandoname : LISTUSPW alias: LU
Parameter : -

Das Kommando löscht den Bildschirm und listet alle zutrittsberechtigten Benutzerkennungen und die entsprechenden passwords.

Kommandoname : REMUSID alias: RU
Parameter : <userid>

Das Kommando entfernt den vorhandenen Eintrag userid aus der Benutzertabelle und löscht den Eintrag des dazugehörenden passwords aus der password-Tabelle. Alle Privat-Dateien von userid werden gelöscht. Die Rechte-Tabellen aller Privat- und System-Dateien werden aktualisiert.

4.1.4.1.2. Allgemeine Kommandos

Auf den SYSTEM-Level gibt es 4 Typen von allgemeinen Kommandos. Diese sind:

1. Dateiverwaltungs-Kommandos
2. Kommandos zur Verwaltung der Rechte-Tabellen
3. Kommandos zum Verlassen der SYSTEM-Level
4. Service-Kommandos

Dateiverwaltungs-Kommandos

Kommandoname : COPYF alias: CF
Parameter : <fileid1> <fileid2>

---

fileid1 bezeichnet eine Datei, auf die der Benutzer Lese- oder Schreibrecht besitzt. Ist fileid1 eine fremde Datei, so ist ein vollqualifizierter Dateiname in Form vom userid.filename anzugeben. So sind auch Systemdateien, die kopiert werden sollen, mit dem Präfix SYS. zu verstehen. fileid2 bezeichnet einen Dateinamen, der noch nicht im Dateikatalog des Benutzers vermerkt sein darf. COPYF kopiert den gesamten Inhalt von fileid1 nach fileid2. Dabei wird das environment von fileid1 auch zu dem von fileid2. Der Benutzer wird Eigentümer von fileid2.

Kommandoname : DELETEF alias: DELF DF
Parameter : <fileid>

---

fileid und die entspechenden Rechte-Tabellen werden gelöscht. Der Benutzer muß Eigentümer von fileid sein.

Kommandos zur Verwaltung der Rechte-Listen

Kommandoname : ENTRYR alias: ER
Parameter : <userid>|ALL <fileid>|ALL RO|RW

Mit diesem Kommando können entweder Rechte auf Dateien an fremde Benutzer erstmalig vergeben werden, oder aber existierende Rechte fremder Benutzer geändert werden. userid bezeichnet einen fremden Benutzer, fileid bezeichnet eine Datei des Benutzers. ALL steht im ersten Fall für alle fremden Benutzer, im zweiten Fall für alle Dateien des Benutzers. RO bezeichnet das Lese- und RW das Schreibrecht. Das Schreibrecht des Benutzers und des Systemmanagers können nicht verändert werden.

Kommandoname : LISTR alias: LR
Parameter : <userid>

Das Kommando löscht den Bildschirm und gibt in Abhängigkeit von userid folgendes listing aus:

1. Gibt der Benutzer für userid seine eigene Benutzerkennung an, so listet das Kommando für alle Dateien von userid die Rechte, die fremde Benutzer an diesen Dateien besitzen.
2. Gibt der Benutzer für userid nicht seine eigene Benutzerkennung an, so listet das Kommando die Dateien, sowie das entsprechende Lese- oder Schreibrecht, das userid dem Benutzer zur Verfügung gestellt hat.

Kommandoname : REMOVER alias: RR
Parameter : <userid>|ALL <fileid>|All

In den Rechtetabellen der Datei fileid oder aller Dateien des aufrufenden Benutzers wird das bestehende Recht von userid oder die Rechte aller Benutzer, falls vorhanden, gelöscht. Hierbei kann ein Benutzer weder sein eigenes Schreibrecht, noch das des System-Managers löschen.

Kommandos zum Verlassen des SYSTEM-Levels

Kommandoname : OPEN
Parameter : <fileid> [<SYS.fileid$_1$>...<SYS.fileid$_n$>]

Das Kommando dient zum Betreten des FILE-Levels. Die Kommandos des SYSTEM-Levels sind nicht mehr verfügbar. Die nun verfügbaren Kommandos beziehen sich auf die Spezifikationen von fileid und die aus dem environment von fileid.

1. fileid bezeichnet eine noch nicht im Dateikatalog des Benutzers vorhandene Datei:
   In diesem Fall wird eine zunächst leere Datei mit diesem Namen generiert. Der Benutzer hat für diese Datei mit SYS.fileid$_1$... SYS.fileid$_n$ das environment dieser Datei zu bestimmen. Das environment besteht aus der Vereinigung der transitiven Hüllen von SYS.fileid$_1$ ... SYS.fileid$_n$. Dem Benutzer stehen die Spezifikationen des environments zur Verfügung. Bezeichnet SYS.fileid$_i$ (i=1...n) keine existierende Systemdatei, wird das Kommando nicht ausgeführt, und eine entsprechende Fehlermeldung ausgegeben.
2. fileid bezeichnet eine im Dateikatalog des Benutzers vorhandene Datei:
   In diesem Fall entfällt die Angabe des environments. Die auf fileid befindlichen Spezifikationen und die aus dem environment werden dem Benutzer zur Verfügung gestellt. Für die Spezifikationen von fileid werden die standardmäßig vorhandenen Operationen erzeugt.

Kommandoname : EXIT
Parameter : -

Die SPESY-Sitzung wird beendet und der Benutzer gelangt zurück auf die Betriebssystem-Ebene der BS2000. Wird bei Ausführung des Kommandos ein Protokoll geführt, so wird dieses beendet. Wurde zu einer beliebigen Zeit während der Sitzung protokolliert, erfragt das Kommando, ob ein Ausdrucken des Protokolls gewünscht wird. Die Eingabe von Y oder YES veranlaßt den Ausdruck und ein anschließendes Löschen der Protokoll-Datei. Eine anders lautende Antwort stellt dem Benutzer das Protokoll in der Datei Benutzerkennung.protocol zur freien Verfügung. Diese Datei ist über SPESY -Kommandos nicht mehr verfügbar. Achtung: Um das System nicht unnötig aufzublähen, wird in der nächsten Sitzung desselben Benutzers, in der ein Protokoll geführt wird, die alte Protokoll-Datei, soweit noch vorhanden, überschrieben.

Service-Kommandos

Kommandoname : DATE alias: D
Parameter : -

Es erscheint folgende Meldung auf dem Bildschirm:

DATE : Datum

Kommandoname : HELP alias: H ?
Parameter : [cmd]

Das Kommando löscht den Bildschirm und listet die auf dem System -Level verfügbaren Kommandos und deren alias-Namen. Wird cmd angegeben, wird entsprechend Information zum Kommando cmd gegeben.

Kommandoname : HELPTAST alias: HTAST HTA
Parameter : -

Das Kommando löscht den Bildschirm und listet die Belegung der Funktionstasten des Terminal-Keyboards auf den Bildschirm.

Kommandoname : LISTALLSPECS alias: LA
Parameter : OWN|SYS|ALL

Das Kommando löscht den Bildschirm und listet entweder die Spezifikationsnamen aus den Privat-Dateien des Benutzers nach Dateien und Typen getrennt oder die Spezifikationsnamen aller System-Dateien oder beides nach Dateien und Spezifikationstypen getrennt.

Kommandoname : LISTC alias: LC
Parameter : -

Das Kommando löscht den Bildschirm und listet den Dateikatalog des Benutzers auf.

Kommandoname : LISTSPECSOF alias: LO
Parameter : <fileid>

Das Kommando löscht den Bildschirm und listet die Spezifikationen aus der Datei fileid.

Kommandoname : PRINT
Parameter : <fileid>

Die auf fileid befindlichen Spezifikationen werden in die Datei Benutzerkennung.PRINTFILE.Versionsnummer geschrieben und danach ausgedruckt. Der Benutzer muß das Leserecht auf fileid besitzen. (Siehe auch PRINT-Kommando in 4.1.4.2.)

Kommandoname : PROTOCOL alias: PROT P
Parameter : HELP|ON|ON?|OFF|EXT|

Das Kommando leistet in Abhängigkeit des Parameters folgendes:

1. HELP : Die Parameter des Kommandos werden erläutert.
2. ON : Das Kommando generiert eine Datei mit dem Namen Benutzerkennung.protocol. Gibt es bereits eine Datei mit diesem Namen aus einer früheren SPESY-Sitzung, so wird diese überschrieben, um ein Aufblähen des Systems zu vermeiden. Jede Eingabe des Benutzers und jede Ausgabe des Systems werden ab diesem Zeitpunkt in der Datei protokolliert.
3. ON? : Der Benutzer wird darüber informiert, ob gerade ein Protokoll geführt wird, oder nicht.
4. OFF : Das Protokollieren wird abgebrochen.
5. EXT : Ein während der Sitzung bereits geführtes und daraufhin abgebrochenes Protokoll wird fortgesetzt.

Bei Verlassen des Systems durch EXIT wird ein laufendes Protokoll automatisch beendet. Der Benutzer hat die Möglichkeit, sich das während der Sitzung angelegte Protokoll durch SPESY ausdrucken zu lassen; in Anschluß an das Drucken wird die Protokoll-Datei automatisch gelöscht. Verzichtet der Benutzer auf den Ausdruck, steht ihm das Protokoll in der angegebenen Datei frei zur Verfügung.

Kommandoname : TIME alias: T
Parameter : -

Es erscheint folgende Meldung auf dem Bildschirm:

TIME : Uhrzeit

Kommandoname : TIMEDATE alias: TD
Parameter : -

Das Kommando ist eine Kombination von TIME und DATE und gibt die Uhrzeit und das Datum an.

4.1.4.2. FILE-Level-Kommando

Auf dem FILE-Level gibt es 5 Typen von Kommandos. Diese sind:

1. LIST/PRINT-Kommandos
2. Kommando zum Löschen von Spezifikationen
3. Testkommandos
4. Kommandos zum Verlassen des FILE-Levels
5. Servicekommandos

Alle Kommandos beziehen sich auf die verfügbaren Spezifikationen. Verfügbar sind die Spezifikationen aus der geöffneten Privat-Datei des Benutzers und die Spezifikationen aus deren environment.

Kommandoname : LINTERFACE alias: LI
Parameter : <specid> SORTS|OPS|ALL

Das Kommando löscht den Bildschirm und listet das downwards-interface von specid. Um das Kommando sinnvoll ausführen zu können, darf sich in der use-Relation von specid kein Zyklus befinden. Sollte ein Zyklus existieren, wird eine entsprechende Fehlermeldung ausgegeben und ein Listing unterdrückt. Der zweite Parameter legt fest, ob die in der Spezifikation zur Verfügung stehenden Sortensymbole, Operationssymbole oder beide gelistet werden sollen. Als Nebenprodukt der Interface-Bestimmung erkennt das Kommando, ob benutzte pspec-terms oder parm-declarations fehlerhaft sind. In diesem Fall wird zum interface-listing eine entsprechende Warnung ausgegeben. In fehlerhaften pspec-terms werden darüberhinaus fehlende oder überflüssige Variablen durch das Symbol # ersetzt.

Kommandoname : LIST
Parameter : <specid>

Das Kommando löscht den Bildschirm und listet die Spezifikation specid auf den Bildschirm.

Kommandoname : LISTP alias: LP
Parameter : <specid> <part>

Das Kommando löscht den Bildschirm und listet den durch part gekennzeichneten Teil der Spezifikation specid. part bezeichnet einen Knoten oder ein Blatt im Hierarchie-Baum einer Spezifikation und muß ein identifier aus dem SPESY-Kürzelkatalog sein.

Kommandoname : LSPUSEDBY alias: LY
Parameter : <specid> D|U|ALL

Das Kommando löscht den Bildschirm und listet alle von der Spezifikation specid direkt oder indirekt benutzten definierten, undefinierten oder alle Spezifikationen. Als Nebenprodukt erkennt das Kommando, ob über die use-relation erreichbare parm-declarations fehlerhaft sind. In diesem Fall wird eine entsprechende Warnung ausgegeben.

Kommandoname : LSPUSING alias: LG
Parameter : <specid>

Das Kommando löscht den Bildschirm und listet die Namen aller Spezifikationen, die die Spezifikation specid benutzen.

Kommandoname : PRINT
Parameter : <specid>|OWN|SYS|ALL

Das Kommando schreibt specid, alle Spezifikationen aus der geöffneten Privat-Datei, alle Spezifikationen aus dem environment oder alle verfügbaren Spezifikationen auf die Datei Benutzerkennung.PRINTFILE.Versionsnummer. Diese wird beim Verlassen des Systems mit EXIT automatisch ausgedruckt und anschließend gelöscht. Sollte ein Benutzer das System und jeweils das PRINT-Kommando in kurzer Folge mehrmals benutzen, kann es vorkommen, daß ältere print-jobs noch nicht vom Betriebssystem abgearbeitet wurden. In diesem Fall werden die alten printfiles nicht von SPESY überschrieben, sondern es wird ein neuer printfile mit einer um 1 höheren Versionsnummer generiert.

Kommando zum Löschen von Spezifikationen

Kommandoname : DELETES alias: DELS DS
Parameter : <specid>

Das Kommando entfernt specid aus der geöffneten Privat-Datei des Benutzers und modifiziert den controlvector jeder Spezifikation, die diese benutzte.

Testkommandos

Kommandoname : ANYCYCLES?
Parameter : -

Das Kommando überprüft für jede Spezifikation aus der geöffneten Privat-Datei, ob in deren use-Beziehungen Zyklen auftreten und listet diese gegebenenfalls.

Kommandoname : CHECK
Parameter : <specid>

Mit diesem Kommando kann eine Spezifikation nach dem Editieren auf syntaktische Korrektheit hin überprüft werden.
Der Kommando-Aufruf kann verschiedene Wirkung haben:

1. Wurde specid bereits mit CHECK überprüft und erfolgte seit der letzten Überprüfung keine Änderung in der durch specid definierten Hierarchie, wird das Kommando nicht ausgeführt und es erfolgt eine entsprechende Meldung.
   Änderungen sind:
   Anwendungen der Kommandos ADD, CHANGE, DELETE im Editor und Verlassen des Editors mit END S (vgl.[KRST 83-1]), sowie Löschen von Spezifikationen, die von specid direkt oder indirekt benutzt werden.
2. Wurden bei der Eingabe von specid bereits die syntaktischen checks vorgenommen und traten keine Fehler auf, so wird das Kommando ebenfalls nicht ausgeführt.
3. Wurde an specid eine Veränderung vorgenommen, so wird die Spezifikation geprüft. Für jede Klausel wird entweder angegeben, daß sie korrekt ist, oder eine entsprechende Fehlermeldung.

Kommandoname : CYCLES?
Parameter : <specid>

Das Kommando überprüft für specid, ob in deren use-Beziehung ein Zyklus auftritt und listet diesen gegebenenfalls.

Kommandos zum Verlassen des FILE-Levels

Kommandoname : CLOSE
Parameter : -

Das Kommando schließt die geöffnete Privat-Datei des Benutzers und bringt den Benutzer zurück auf den SYSTEM-Level, dessen Kommandos nun wieder verfügbar sind. Alle an der Datei und den Spezifikationen der Datei vorgenommenen Änderungen werden gerettet durch Überschreiben der alten Dateiversion. Die Kommandos des FILE-Levels sind nicht mehr verfügbar.

Kommandoname : EDIT
Parameter : <specid>

Mit dem Kommando wird der FILE-Level verlassen und der EDIT-Level betreten. Die Kommandos des FILE-Levels sind nicht mehr verfügbar.

Anmerkung: Die Implementierung des Kommandos war nicht Gegenstand dieser Arbeit. Details sind zu finden in [KRST 83-1].

Kommandoname : INPUT alias: DEFINE
Parameter : -

Mit dem Kommando wird der FILE-Level verlassen und der INPUT-Level betreten. Die Kommandos des FILE-Levels sind nicht mehr verfügbar.
Anmerkung: Die Implementierung des Kommandos war nicht Gegenstand dieser Arbeit. Details sind zu finden in [KRST 83-1].

Kommandoname : INSTANTIATE alias: INST
Parameter : -

Mit dem Kommando wird der FILE-Level verlassen und der INSTANTIATE-Level betreten. Die Kommandos des FILE-Levels sind nicht mehr verfügbar.
Anmerkung: Eine detaillierte Beschreibung des Kommandos erfolgt in Abschnitt 4.1.4.3.

Service-Kommandos

Kommandoname : DATE alias: D
Parameter : -

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1.

Kommandoname : FLOADSET alias: FLS
Parameter : -

Das Kommando löscht den Bildschirm und listet alle verfügbaren Spezifikationsnamen, unterschieden nach Spezifikationstypen.

Kommandoname : HELP alias: H ?
Parameter : [cmd]

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1.

Kommandoname : HELPABREVIATION alias: HA
Parameter : SSPEC|PSPEC|PARM

Das Kommando löscht den Bildschirm und listet den Kürzelkatalog für den entsprechenden Spezifikationstyp, deren Bedeutung und die jedem Teil zugeordnete Nummernfolge, über die man den Teil im Strukturbaum einer Spezifikation erreicht.

Kommandoname : HELPTAST alias: HTAST HTA
Parameter : -

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1

Kommandoname : HELPTREE alias: HT
Parameter : SSPEC|PSPEC|PARM

Das Kommando listet eine graphische Darstellung des entsprechenden Spezifikationstyps in der allgemeinen Baumstruktur.

Kommandoname : LISTALLSPECS alias: LA
Parameter : OWN|SYS|ALL

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1

Kommandoname : LISTSPECSOF alias: LO
Parameter : <fileid>

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1

Kommandoname : LNOKSPECS alias: LNOK
Parameter : -

Das Kommando löscht den Bildschirm und listet die Namen aller Spezifikationen, deren Komponenten nicht vollständig ok sind.

Kommandoname : LOADSET alias: LS
Parameter : -

Das Kommando löscht den Bildschirm und listet die Namen aller Spezifikationen aus der geöffneten Privat-Datei nach Spezifikationstypen getrennt.

Kommandoname : LOKSPECS alias: LOK
Parameter : -

Das Kommando löscht den Bildschirm und listet die Namen aller Spezifikationen, deren Komponenten syntaktisch vollständig korrekt sind.

Kommandoname : PROTOCOL alias: PROT P
Parameter : HELP|ON|ON?|OFF|EXT

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1

Kommandoname : TIME alias: T
Parameter : -

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1

Kommandoname : TIMEDATE alias: TD
Parameter : -

Kommandobeschreibung siehe SYSTEM-Level 4.1.4.1

4.1.4.3. INSTANTIATION-Level

Der INSTANTIATION-Level wird vom FILE-Level aus betreten. Die Kommandos des FILE-Levels sind nicht mehr verfügbar. Im Gegensatz zu SYSTEM- und FILE-Level gibt es nur ein Kommando. Der Kommando-Aufruf kann in einer der folgenden Formen geschehen:

1. INSTANTIATE | INST
2. INSTANTIATE | INST <specidl>
3. INSTANTIATE | INST <specidl> TO
4. INSTANTIATE | INST <specidl> TO <specid2>

Bei den unvollständigen Kommando-Aufrufen 1. - 3. fordert das Kommando die fehlende Eingabe an. Es ergänzt den Aufruf im ersten und zweiten Fall jeweils um das Schlüsselwort to und fordert, falls nötig, specid2 an.

Anmerkung: Um die zum Kommando gehörenden Schlüsselwörter auf dem Bildschirm besser von den Benutzereingaben oder Teilen von Spezifikationen wie Sorten- und Operationssymbole, pspec-terms, parm-variables usw. unterscheiden zu können, werden die Schlüsselwörter vom System durchgehend in Klein-Buchstaben, alle anderen Symbole in Groß-Buchstaben auf den Bildschirm geschrieben.

specidl muß eine pspec oder einen zusammengesetzten parm aus den verfügbaren Spezifikationen bezeichnen. Sowohl der spec-header, als auch die parm-declaration von specidl müssen syntaktisch korrekt sein. Ist eine der drei Bedingungen nicht erfüllt, bricht das Kommando ab, eine entsprechende Fehlermeldung wird ausgegeben, und der Benutzer gelangt zurück auf den FILE-Level.

specid2 muß ein gültiger Spezifikationsname sein (siehe 4.1.5.) und darf keine bereits verfügbare Spezifikation bezeichnen. Im Falle eines Fehlers wird der Benutzer aufgefordert, einen neuen Spezifikationsnamen einzugeben. Das wiederholt sich so lange, bis entweder ein korrekter Name eingegeben wurde, oder aber der Benutzer durch Drücken der return-Taste das Kommando abbrechen möchte. In diesem Fall gelangt der Benutzer zurück auf den FILE-Level.

Ist der Kommando-Aufruf fehlerfrei abgearbeitet worden, erscheint das prompt-Symbol **declare**, und der Benutzer hat die Möglichkeit die actual-parm-declaration einzugeben. Drei Eingaben sind möglich:

1. Drücken der return-Taste
   Die leere Eingabe wird dahingehend interpretiert, daß der Benutzer auf die actual-parm-declaration verzichtet. Es erscheint das prompt-Symbol **actualize**.
2. Help | H | ?
   Der Benutzer wird aufgefordert die actual-parm-declaration einzugeben oder die return-Taste zu betätigen.
3. actual-parm-declaration
   1) Die Eingabe der actual-parm-declaration kann strukuriert in mehreren Schritten erfolgen. Dabei dürfen zwischen den syntaktischen Einheiten wie Variablen, specids, Kommata, Doppelpunkten und Klammern beliebig viele blanks stehen. Die Eingabe wird als beendet angesehen, wenn die erste öffnende Klammer der parm-declaration wieder geschlossen wurde.
   Die Eingabe wird zunächst auf syntaktische Korrektheit hin überprüft. Im Falle eines Fehlers wird eine ausführliche Fehleranalyse ausgegeben, in der die fehlerhaften Teile durch das Symbol *ERROR* ersetzt sind.

Beispiel:
Ein Benutzer gibt folgende actual-parm-declaration ein:

```
(a:Arrayspec(#e:Elem.pec,
             #lim1:Limitspec(#ind1:Indexspec),
             #lim2:Limitspec(#ind1:Indexspec))
```

dann erhält er folgende Diagnose:

```
INVALID PARAMETER-DECLARATION.
INVALID PARAMETER-VARIABLE: a.
INVALID SPECNAME: ELEM.PEC
PARAMETER-VARIABLE #IND1 TWICE DEFINED.
(*ERROR*:ARRAYSPEC(#E:*ERROR*,
                   #LIM1:LIMITSPEC(#IND1:INDEXSPEC),
                   #LIM2:LIMITSPEC(*ERROR*:INDEXSPEC)).
                                                      •
```

2) Der Benutzer wird aufgefordert, eine neue actual-parm-declaration einzugeben oder aber durch Betätigen der return-Taste das Kommando abzubrechen.

   Sind bei der syntaktischen Überprüfung keine Fehler aufgetreten, wird geprüft, ob die parm-declaration wohl definiert ist. Im Falle eines Fehlers erfolgt eine ausführliche Fehleranalyse. Der Benutzer wird dann wiederum aufgefordert, eine parm-declaration einzugeben, oder aber durch Betätigen der return-Taste das Kommando abzubrechen.

3) Sind in den ersten beiden Stufen keine Fehler aufgetreten, wird anschließend überprüft, ob sämtliche in actual-parm-declaration eingeführten parm-Variablen verschieden sind von denen aus der parm-declaration der Spezifikation, die instanziiert wird. Im Falle eines Fehlers erfolgt eine Auflistung der wiederholt vorkommenden Variablen und die Aufforderung:
   REENTER ACTUAL-PARM-DECLARATION.
   Wurde die actual-parm-declaration vom System akzeptiert, erscheint das prompt-Symbol actualize.

In der Aktualisierungsphase verarbeitet das System drei Arten von Benutzereingaben:

1. Drücken der return-Taste
   Das System untersucht, ob die Aktualisierungen vollständig waren, d.h. wurden für jeden formalen Parameter f_i, der aktualisiert wurde, auch all die formalen Parameter aktualisiert, die f_i benutzen. Ist das der Fall, erscheint das prompt-Symbol **rename sorts** und die Aktualisierungsphase ist abgeschlossen. Im anderen Fall erscheint die Warnung:

   WARNING: $f_i \ldots f_n$
   USING ACTUALIZED PARAMETERS IS/ARE NOT ACTUALIZED.
   DO YOU WANT TO TERMINATE THE ACTUALIZATION?,

   wobei f_i i=1...n formale Parameter sind, die noch aktualisiert werden müssen, damit die Instanziierung korrekt ist.
   Die Eingabe von N oder NO bricht die Aktualisierungsphase nicht ab. Das System erwartet die Eingabe der nächsten Aktualisierung.
   Die Eingabe von Y oder YES bricht das Instanziierungs-Kommando ab. Der Benutzer gelangt zurück auf den FILE-Level.
   Jede andere Antwort hat:

   INVALID ANSWER - ANSWER: `Y´ OR `N´

   zur Folge. Das System erwartet eine entsprechende Eingabe.

2. HELP | H | ?
   Es erscheint folgende Meldung:

   ENTER ACTUALIZATION:
   FORMAL-PARAMETER `BY | BY_MATCH | BY_INSTANCE´ ACTUAL PARAMETER
   OR >RET< TO FINISH ACTUALIZATION.

3. actualization

Die einzelnen Aktualisierungen müssen vollständig eingegeben werden in der Form: formal$^+$ Aktualisierungs-Typ actual

An formal$_i$ i=1...n werden folgende Bedingungen gestellt:

1. formal$_i$ darf nur einmal unter formal$_1$...formal$_n$ aufgeführt sein.
2. formal$_i$ darf noch nicht aktualisiert worden sein.
3. formal$_2$...formal$_n$ müssen vom gleichen Typ sein, wie formal$_1$.
4. formal$_i$ muß ein pspec-term oder eine parm-variable sein, und formal$_i$ wird von der Spezifikation, die instanziiert wird, benutzt.
5. Wurden formale Parameter, die von formal$_i$ benutzt werden, bereits früher aktualisiert, so müssen beide Aktualisierungen übereinstimmen.
6. Alle von formal$_i$ benutzten Parameter müssen bereits aktualisiert sein; die Aktualisierung muß das Hierarchie-requirement erfüllen.

Ist eine der Bedingungen für formal$_i$ nicht erfüllt, wird die Aktualisierung für formal$_i$ zurückgewiesen. Im Anschluß an die Aktualisierung der restlichen Variablen wird eine entsprechende Fehlermeldung ausgegeben.

3.1. formal$^+$ by_match actual

1. Ist formal$_i$ i=1...n eine parm-variable, dann muß actual eine parm-variable aus der actual-parm-declaration sein. Der Typ der formalen Parameter und von actual müssen identisch sein. Sonst wird die Aktualisierung zurückgewiesen und eine entsprechende Fehlermeldung ausgegeben.
2. Ist formal$_i$ i=...n ein pspec-term, dann muß auch actual ein pspec-term sein, dessen parm-variablen aus der actual-parm-declaration stammen. actual muß in Bezug auf alle verfügbaren Spezifikationen wohl definiert sein. Der Typ der formalen Parameter und von actual müssen identisch sein.

Ansonsten wird die Aktualisierung zurückgewiesen und eine entsprechende Fehlermeldung ausgegeben.

3.2. formal^{+} by_instance actual

1. Sei $formal_i$ i=1...n eine parm-variable vom Typ FP, dann muß actual eine parm-variable aus actual-parm-declaration sein, deren Typ einen parm bezeichnet, der eine Instanz ist von FP. Sonst wird die Aktualisierung zurückgewiesen und eine entsprechende Fehlermeldung ausgegeben.
2. Sei $formal_i$ i=1...n ein pspec-term vom Typ FP, dann gilt für actual:
   1. actual bezeichnet eine sspec, die eine Instanz ist von FP.
   2. actual ist ein pspec-term, der eine pspec bezeichnet, die eine Instanz ist von FP. Die parm-variablen von actual stammen aus der actual-parm-declaration. actual ist wohl definiert bezüglich der verfügbaren Spezifikationen.

   Ist eine der Bedingungen nicht erfüllt, wird die Aktualisierung zurückgewiesen und eine entsprechende Fehlermeldung ausgegeben.

3.3. formal^{+} by actual

Für actual gilt:

1. Ist $formal_i$ i=1...n eine parm-variable, dann darf actual kein pspec-term sein, da sonst die use-Beziehung zwischen den verschiedenen Spezifikationstypen verletzt würde.
2. Ist actual ein pspec-term, dann müssen die parm-Variablen aus der actual-parm-declaration stammen und actual muß wohl definiert sein bezüglich der verfügbaren Spezifikationen.
3. Ist actual weder eine parm-variable noch ein pspec-term, dann muß es eine verfügbare sspec bezeichnen.

Ist eine der Bedingungen nicht erfüllt, wird die Aktualisierung abgebrochen und eine entsprechende Fehlermeldung ausgegeben. Im anderen Fall erfragt das System für jedes

Sorten- bzw. Operationssymbol f_i der formalen Parameter das aktuelle Sorten- bzw. Operationssymbol a_i aus dem upwards-interface von actual:

1. Aktualisieren der public sorts der formalen Parameter

1.1. a_i bezeichnet eine Sorte aus dem upwards-interface von actual und ist eindeutig. Die Eingabe wird akzeptiert und das prompting fortgesetzt. Nach Aktualisierung aller Sortensymbole erfolgt die Aktualisierung der Operationssymbole.

1.2. a_i bezeichnet eine Sorte aus dem upwards-interface von actual und ist mehrdeutig, dann erfolgt die Aufforderung:

QUALIFY a_i :

Folgende Eingaben sind möglich:

1.2.1. Gibt der Benutzer HELP, H oder ? ein, so erscheint die Meldung:

ENTER NAME OF SPEC WHERE a_i IS DEFINED OR `STOP´.

und das System erwartet eine entsprechende Eingabe.

1.2.2. Antwortet der Benutzer mit STOP, wird die laufende Aktualisierung abgebrochen und ignoriert. Der Benutzer kann eine weitere Aktualisierung eingeben, oder aber die abgebrochene neu beginnen.

1.2.3. Gibt der Benutzer einen sspec-Namen, pspec-term oder eine parm-variable als Qualifikation ein, so muß actual die entsprechende Spezifikation benutzen, und a_i muß aus deren public sorts stammen. Ist dies nicht der Fall, wird eine entsprechende Fehlermeldung ausgegeben und das prompting der letzten Sorte wird wiederholt.

1.3. a_i hat die Form pref.s:

1.3.1. Bezeichnet pref eine von actual benutzte Spezifikation, und ist s eine public sort von pref, dann wird die Eingabe akzeptiert.

1.3.2. Wird pref nicht von actual benutzt, oder ist s keine public sort von pref, wird eine entsprechende Fehlermeldung ausgegeben und das prompting der letzten Sorte wiederholt.

1.4. Bezeichnet a_i keine Sorte aus dem upwards-interface von actual, wird eine entsprechende Fehlermeldung ausgegeben und das prompting der letzten Sorte wiederholt.

1.5. Ist a_i das Symbol STOP, wird die laufende Aktualisierung abgebrochen und ignoriert. Der Benutzer kann weitere Aktualisierungen eingeben, oder aber die abgebrochene neu beginnen.

1.6. Auf die Eingabe von HELP, H oder ? folgt:

ENTER ACTUAL SORTID OR `STOP´.

2. Aktualisieren der public ops

2.1. a_i bezeichnet eine Operation aus dem upwards-interface von actual und ist eindeutig.

1. alle formalen Sorten aus f_i werden ersetzt durch die entsprechenden aktuellen Sorten;
2. alle nicht formalen Sorten aus f_i werden identisch abgebildet.

a_i ist zulässig als aktuelle Operation für f_i, wenn die Stelligkeiten und Zielsorten vor a_i und f_i identisch sind. Im Fall eines Fehlers erscheint die Meldung:

REJECTED : ACTUALISATION IS NO MORPHISM.

und das letzte prompting wird wiederholt.

2.2. a_i bezeichnet eine Operation aus dem upwards-interface von actual und ist mehrdeutig, dann erfolgt die Aufforderung:

QUALIFY a_i :

Folgende Eingaben sind möglich:

2.2.1. Gibt der Benutzer HELP, H oder ? ein, so erscheint die Meldung:

ENTER NAME OF SPEC WHERE a_i IS DEFINED OR `STOP´.

und das System erwartet eine entsprechende Eingabe.

2.2.2. Antwortet der Benutzer mit STOP, wird die laufende Aktualisierung abgebrochen und ignoriert. Der Benutzer kann eine weitere Aktualisierung eingeben, oder aber die abgebrochene neu beginnen.

2.2.3. Gibt der Benutzer einen sspec-Namen, pspec-term oder eine parm-variable als Qualifikation ein, so muß actual die entsprechende Spezifikation benutzen, und a_i muß aus deren public ops stammen. Ist dies nicht der Fall, wird eine entsprechende Fehlermeldung ausgegeben und das prompting der letzten Operation wird wiederholt.

2.3. a_i hat die Form pref.o:

2.3.1. Bezeichnet pref eine von actual benutzte Spezifikation, und ist o eine public operation von pref, dann wird die Eingabe unter der folgenden Bedingung akzeptiert: Stelligkeit und Zielsorte von o und der zu aktualisierenden Oeration müssen nach der in 2.1. beschriebenen Ersetzung der Sortensymbole identisch sein.

2.3.2. Wird pref nicht von actual benutzt, oder ist o keine public operation von pref, wird eine entsprechende Fehlermeldung ausgegeben und das prompting der letzten Operation wiederholt.

2.4. Bezeichnet a_i keine Operation aus dem upwards-interface von actual, wird eine entsprechende Fehlermeldung ausgegeben und das prompting der letzten Operation wiederholt.

2.5. Ist a_i das Symbol STOP, wird die laufende Aktualisierung abgebrochen und ignoriert. Der Benutzer kann weitere Aktualisierungen eingeben, oder aber die abgebrochene neu beginnen.

2.6. Auf die Eingabe von HELP, H oder ? folgt:

ENTER ACTUAL OPID OR `STOP´.

Wurden die Aktualisierungen korrekt beendet, folgt das prompt-Symbol **rename sorts**, soweit in der Spezifikation, die instanziiert wird, public sorts vorhanden sind. Anschließend fordert das System für jedes vorhandene Sortensymbol einen neuen Namen an. Folgende Eingaben sind möglich:

1. Drücken der return-Taste bedeutet, daß für das alte Symbol kein neues eingeführt werden soll.
2. Die Eingabe von ;; beendet das prompting der Sortensymbole. Die bereits vorgenommenen Umbenennungen bleiben erhalten. Das System fährt mit dem prompting der Operationssymbole fort.
3. Die Eingabe von HELP, H oder ? bewirkt den Ausdruck:

ENTER NEW SORTID OR >RET< OR `;;´ TO STOP RENAMING Of SORTS:

Das System erwartet eine entsprechende Eingabe.

4. Wird ein Sortensymbol s eingegeben, so darf dieses noch nicht unter den neubenannten und nicht umbenannten Sortensymbolen vorhanden sein. Außerdem muß s ein gültiger Sortenname sein (siehe 4.1.5.). Im Fall eines Fehlers wird eine entsprechende Meldung ausgegeben und das prompting der letzten Sorte wiederholt.

Nach Umbenennung der Sortensymbole erfolgt analog die Umbenennung der Operationssymbole. Das System erfragt wiederum für jedes Operationssymbol den neuen Namen. Folgende Eingaben sind möglich:

1. Drücken der return-Taste bedeutet, daß für das alte Symbol kein neues eingeführt werden soll.
2. Die Eingabe von ;; beendet das prompting der Operationssymbole. Die bereits vorgenommenen Umbenennungen bleiben erhalten. Das Kommando ist damit beendet.
3. Die Eingabe von HELP, H oder ? bewirkt den Ausdruck:

   ENTER NEW OPID OR >RET< OR `;;´ TO STOP RENAMING Of OPS:

   Das System erwartet eine entsprechende Eingabe.
4. Wird ein Operationssymbol o eingegeben, so gelten folgende Bedingungen:

4.1. o darf noch nicht unter den umbenannten und nicht umbenannten Sortensymbolen vorkommen;

4.2. o darf nicht unter den bereits umbenannten oder nicht umbenannten Operationssymbolen vorkommen;

4.3. o darf nicht als private ops in der zu instanziierenden Spezifikation vorkommen;

4.4. o muß ein gültiger Operationsname sein (siehe 4.1.5.).

Ist das renaming beendet, erstellt das System eine neue Spezifikation als Instanz der instanziierten Spezifikation. Es erfolgt die Ausgabe:

INSTANCE <specid> OF <specid2> GENERATED.

Der Benutzer gelangt zurück auf den FILE-Level und die Instanz

wird in der geöffneten Privat-Datei aufgenommen.

Anmerkung: Wird bei der Instanziierung keine Aktualisierung durchgeführt, dann kann das Kommando benutzt werden um eine Kopie einer Spezifikation zu erstellen, bei der, dem renaming entsprechend, Sorten oder Operationen umbenannt sind. Dabei werden alle zu einem Sorten- oder Operationssymbol korrespondierenden Symbole in der Kopie neu generiert und ersetzt.

4.1.4.4. EDIT-Level INPUT-Level

Die Implementierung des EDIT- und INPUT-Kommandos waren nicht Gegenstand dieser Arbeit. Detaillierte Informationen über diese Kommandos sind zu finden in [KRST 83-1].

4.1.5. Namenskonventionen

4.1.5.1. Allgemeine Namenskonventionen

1. Dateinamen
   - Ein Dateiname darf höchstens 30 Zeichen lang sein.
   - Ein Dateiname muß aus alphanumerischen Zeichenfolgen bestehen.

2. Spezifikationsnamen
   - Ein Spezifikationsname darf höchstens 50 Zeichen lang sein.
   - Er hat folgenden Aufbau:
     <einfacher Name> |
     <einfacher Name>(<Spezifikationsname>$_1$,...,<Spezifikationsname>$_n$).
   - Ein einfacher Name beginnt mit einem Buchstaben und enthält sonst nur alphanumerische Zeichen.

3. Sortennamen
   - Ein Sortenname ist ein einfacher Name, der höchstens 16 Zeichen lang ist.

4. Parameter-Variablen
   - Eine parm-Variable darf höchstens 17 Zeichen lang sein.
   - Sie ist ein einfacher Name, der weder Komma noch Doppelpunkt enthält und mit einem # beginnt.

5. Constructor-Namen
   - Ein constructor-Name ist maximal 17 Zeichen lang und beginnt mit einem *.
   - Der Name ohne den führenden * muß ein Operationsname sein.

6. Auxiliary-Namen
   - Ein auxiliary-Name ist maximal 17 Zeichen lang und beginnt mit einem $.
   - Der Name ohne den führenden $ muß ein Operationsname sein.

7. Standard-Operationsnamen
   - Ein Standard-Operationsname hat folgenden Aufbau:
     eq.<Sortenname> |
     error.<Sortenname> |
     $eq.<Sortenname> |
     is-<Sortenname> |
     $is-<Sortenname> |
     $arg-<nr>-<Sortenname>,
     wobei <nr> eine natürliche Zahl in arabischer Notation bezeichnet. Standard-Operationsnamen werden automatisch erzeugt.

8. Operationsnamen
   - Ein Operationsname ist maximal 16 Zeichen lang.
   - Besteht der Name aus einem Zeichen, darf dieses beliebig sein.
   - Operationsnamen, die länger als ein Zeichen sind, sind einfache Namen.

9. Variablen-Namen
   - Variablen-Namen sind einfache Namen, die maximal 16 Zeichen lang sind.

4.1.5.2. Eindeutigkeit von Namen

- Spezifikationsnamen sind von den Datei-Namen eines Benutzers verschieden.

- Die in 4.1.5.1. unter den Ziffern 3. bis 9. aufgeführten Namen sind von den Spezifikationsnamen ihrer Umgebung und von den Datei-Namen des Benutzers verschieden.

- Werden in einer Spezifikation zwei gleiche Sorten- oder Operationsnamen benutzt, so werden diese zur Unterscheidung mit dem Spezifikationsnamen, pspec-term oder parm-variable als Präfix versehen, in dessen bzw. deren public-clause sie definiert wurden. Dieses gilt nur für Sorten- oder Operationsnamen aus dem downwards-interface; d.h. Symbole, die in der Spezifikation eingeführt wurden, haben kein Präfix.

4.1.5.3. Spezielle Operationsnamen

- Infix-Operatoren
  Es besteht die Möglichkeit, Infix-Operatoren zu verwenden. Diese haben genau zwei Argumente. Ihre Namen beginnen und enden mit dem Zeichen _ und bestehen damit aus mindestens drei Zeichen. Infix-constructors und -auxiliaries haben die Form:
  *<Operationsname> bzw. _$<Operationsname>_

- Bracket-Operatoren
  Um Vektoren, Tupel, Absolutbetrag u.ä. Operationen in der bekannten Schreibweise verwenden zu können, sind folgende bracket-Operatoren zugelassen:
  () , <> , {} , [] , ||
  Diese können eine beliebige Anzahl von Argumenten haben.

4.1.6. Schlüsselwörter

TOP	TOP
PMDECL	ParM-DECLaration
SPH	SPec-Header
UCL	Use-CLause
SSU	SSpec-Uses
PSU	PSpec-Uses
PUCL	PUbilc-CLause
SOPUCL	SOrts of PUblic-CLause
OPPUCL	OPs of PUblic-CLause
PUDCL	PUblic-Describtive-CLause
EQNS	EQatioNS
INCL	INstantiation-CLause
SPID	SPec-IDentifier (and parm-declaration)
ACCL	ACtualization-CLause
RECL	REname-CLause
SORECL	SOrts of REname-CLause
OPRECL	OPs of REname-CLause
SPB	SPec-Body
CAPA	CArrier-PArt
CCL	Constructor-CLause
AUXCL	AUXiliaries-CLause
OPAUX	OPs of AUXiliaries-CLause
DEFAUX	DEFine-AUXiliaries
DEFCA	DEFine-CArriers
OPPA	OPs-PArt
DEFC	DEFine-Constructors
PROP	PRivate-OPs
DEFOP	DEFine-OPs
PRDCL	PRivate-Descriptive-CLause
DSP	Description of SPec

02/21/1984

SEKI Memos

The following memos are available free of charge from

Mrs. Dorothea Kilgore
Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
West Germany

Memos marked with an asteriks are out of print and only available for a nominal charge.

MEMO SEKI-80-09 Ulrich Guntram: Korrekte Implementierung abstrakter Datentypen durch Moduln in höheren Programmiersprachen.

MEMO SEKI-81-01 U. Bartels, W. Olthoff and P. Raulefs: APE: An Expert System for Automatic Programming from Abstract Specifications of Data Types and Algorithms.

MEMO SEKI-81-03 Peter Raulefs: Expert Systems: State of the Art and Future Prospects.

*MEMO SEKI-81-04 Christoph Beierle: Programmsynthese aus Beispielsfolgen.

MEMO SEKI-81-05 Erich Rome: Implementierungen Abstrakter Datentypen in terminaler Algebrasemantik.

MEMO SEKI-81-06 Christoph Beierle: Synthesizing Minimal Programs from Traces of Observable Behaviour. APE - Benutzerbeschreibung.

*MEMO SEKI-81-07 Dieter Wybranietz: Ein verteiltes Betriebssystem für CSSA.

MEMO SEKI-82-01 Hans Voß: Programming in a Distributed Environment: A Collection of CSSA Examples.

MEMO SEKI-82-02 Hartmut Grieneisen: Eine algebraische Spezifikation des Software-Produkts INTAKT.

MEMO SEKI-82-03 Christian Beilken, Friedemann Mattern and Michael Spenke: Entwurf und Implementierung von CSSA - Beschreibung der Sprache, des Compilers und des Mehrrechnersimulationssystems.
Printed in 6 volumes, which can be ordered individually:
Vol-A: Konzepte
Vol-B: CSSA-Sprachbeschreibung
Vol-C: CSSA-Systembenutzung
Vol-D: CSSA-Programmbeispiele
Vol-E1: Programmdokumentation Teil I
Vol-E2: Programmdokumentation Teil II

MEMO SEKI-83-01 Wilfried Schrupp and Johann Tamme: Spezifikation und abstrakte Implementierung des Aufbereitungsteils von INTAKT.

MEMO SEKI-83-02 Frank Puppe and Bernd Puppe: Overview on MED1: A Heuristic Diagnostic System with an Efficient Control-Structure.

MEMO SEKI-83-03 Elisabeth Hülsmann: LISP-SP : A portable INTERLISP Subset Interpreter for Mini-Computers.

MEMO SEKI-83-04 FrankPuppe: MED1 - Ein heuristisches Diagnosesystem mit effizienter Kontrollstruktur.

MEMO SEKI-83-05 Horst Peter Borrmann: MODIS - Ein Expertensystem zur Erstellung vonReparaturdiagnosen für den Ottomotor und seine Aggregate.

MEMO SEKI-83-06 Harold Boley: From Pattern-Directed to Adapter-Driven Computation via Function-Applying Matching.

MEMO SEKI-83-07 Christoph Beierle and Angi Voß: Canonical Term Functors and Parameterization-by-use for for the Specification of Abstract Data Types.

MEMO SEKI-83-08 Christoph Beierle and Angi Voß: Parameterization-by-use for hierarchically structured objects.

MEMO SEKI-83-09 Christoph Beierle, Michael Gerlach and Angi Voß: Parameterization without parameters in : The History of a Hierarchy of Specifications.

MEMO SEKI-83-10 Ulrike Petersen: Elimination von Rekursionen.

MEMO SEKI-83-11 Gerd Krützer: An Approach to Parameterized Continuous Data Types.

MEMO SEKI-83-12 Richard Göbel: A Completion Procedure for Globally Finite Term Rewriting Systems.

MEMO SEKI-83-13 Michael Gerlach: A Second-Order Matching Procedure for the Practical Use in a Program Transformation System.

MEMO SEKI-83-14 Harold Boley: FIT - PROLOG: A Functional/ Relational Language Comparison. December 1983

MEMO SEKI-84-01 Christoph Thomas: RRLab - Rewrite Rule Labor Entwurf, Spezifikation und Implementierung eines Softwarewerkzeugs zur Erzeugung und Vervollständigung von Rewrite Rule Systemen.

MEMO SEKI-84-02 Walter Sommer: SPESY - Ein Interaktives System zur Unterstützung Integrierter Programm-Spezifikation und -Verifikation

Printed in 3 Volumes. Orders should specify the desired volume(s). Vol. III is only available for a nominal charge.
Vol. I: Description of ASPIK, Examples, SPESY-manual
Vol. II: System documentation
Vol. III: Source code, Session protocols

7. Literaturverzeichnis

[ADJ 76] Goguen,J.A.; Thatcher,J.W.; Wagner,E.G.;
An Initial Approach to the Specification, Correctness, and Implementation of Abstract Data Types,
Research Report IBM, Thomas J. Watson Research Center, Yorktown Heights, New York 1976

[BES 81] Bläsius,K.; Eisinger,N.; Siekmann,J.;
The Markgraf Carl Refutation Procedure,
Proc. 7th IJCAI, 1981

[BGORV 81-1] Beierle,C.; Guntram,U.; Oberdörster,W.; Raulefs,P.; Voß,A.;
Syntax of TRIPLEX, Version 1, A language for systems of algebraic specifications (draft),
Universität Bonn, FB Informatik, Institut III, Dezember 1981

[BGORV 81-2] Beierle,C.; Guntram,U.; Oberdörster,W.; Raulefs,P.; Voß,A.;
Parameterized systems of algebraic specifications in TRIPLEX - an introduction,
Universität Bonn, FB Informatik, Institut III, Dezember 1981

[BGORV 82] Beierle,C.; Guntram,U.; Oberdörster,W.; Raulefs,P.; Voß,A.;
The CTA-approach to the specification of abstract data types,
Universität Bonn, FB Informatik, Institut III, März 1982

[BV 83-1] Beierle,C.; Voß,A.;
Parameterization-by-use for hierarchicaly structured objects,
Universität Kaiserslautern, FB Informatik, SEKI-Projekt, Mai 1983

[BV 83-2] Beierle,C.; Voß,A.;
Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types,
Universität Kaiserslautern, FB Informatik, SEKI-Rrojekt, 1983

[EKP 78] Ehrig,H.; Kreowski,H.-J.; Pradawitz,P.;
Stepwise Specification and Implementation of Abstract Data Types,
TU Berlin, FB Informatik, März 1978

[EKP 79] Ehrig,H.; Kreowski,H.-J.; Pradawitz,P.;
Algebraische Implementierung abstrakter Datentypen,
TU Berlin, FB Informatik, 1979

[GE 83] Gerlach,M.;
A Second-Order Matching Procedure for the Practical Use in a Program Transformation System,
Universität Kaiserslautern, FB Informatik, SEKI-Projekt, 1983

[GR 82] Grieneisen,H.;
Eine algebraische Spezifikation des Software-Produkts INTAKT,
Universität Bonn, FB Informatik, Diplom-Arbeit, 1982

[KL 80] Klaeren,H.;
A simple class of algorithmic specifications of abstract software modules,
Proc. 9th MFCS 1980, LNCS Vol. 88

[KLAE 82] Klaeren,H.A.;
Einführung in die Abstrakte Software-Spezifikation,
TH Aachen, Bericht Nr. 75, Februar 1982

[KREO 79] Kreowski,H.-J.;
Algebra für Informatiker, Schriftliches Material zur gleichnamigen Lehrveranstaltung WS 1978/79
TU Berlin, FB Informatik, 1979

[KRST 83-1] Kücke,R.; Rome,E.; Sommer,W.; Thomas,C.;
Das SPEc-SYstem (SPESY), Benutzerhandbuch,
Universität Kaiserslautern, FB Informatik, SEKI-Projekt, 1983

[KRST 83-2] Kücke,R.; Rome,E.; Sommer,W.; Thomas,C.;
Das SPEc-SYstem (SPESY), Systemdokumentation,
Universität Kaiserslautern, FB Informatik, SEKI-Projekt, 1983

[KO 83] Kücke,R.;
Ein symbolischer Interpretierer für algorithmische ASPIK-Spezifikationen,
Universität Bonn, FB Informatik, Diplom-Arbeit (Stand 1983)

[LO 81-1] Loeckx,J.;
Implementations of Abstract Data Types and Their Verification,
Universität des Saarlandes, FB 10-Informatik, Saarbrücken, interner Bericht, April 1981

[LO 81-2] Loeckx,J.;
A new specification method for abstract data types,
Universität des Saarlandes, FB 10-Informatik, Saarbrücken, interner Bericht, Dezember 1981

[OLT 83] Olthoff,W.;
On the connection of abstract specifications and concrete programs,
Universität Kaiserslautern, FB Informatik, SEKI-Projekt, 1983

[RAUL 79] Raulefs,P.;
Einführung in die Theorie der Datenstrukturen,
Universität Bonn, FB Informatik, Vorlesungsnotizen, SS 1979

[RS 80] Raulefs,P.; Siekmann,J.;
Programmverifikation, Darstellung des Forschungvorhabens,
Universität Bonn, Universität Karlsruhe, August 1980

[RS 82] Raulefs,P.; Siekmann,J.;
Entwicklung eines integrierten Softwareentwicklungs- und Programmverifikationssystems, Vorhabensbeschreibung und Planungsunterlagen,
Universität Bonn, Universität Karlsruhe, Juni 1982

[SEKI 83] Beierle,C.; Gerlach,M.; Göbel,R.; Olthoff,W.; Raulefs,P.; Rome,E.; Sommer,W.; Thomas,C.; Urbassek, C.; Voß,A.;
Integrated Program Development and Verification,
Universität Kaiserslautern, FB Informatik, SEKI-Projekt, PRELIMINARY DRAFT, 1983

[SIEM 80] SIEMENS Arbeitsgruppe INTERLISP
SIEMENS-INTERLISP, Benutzerhandbuch, Version 4.0
SIEMENS München, Vorläufige Ausgabe, August 1980

[SIEM 80] SIEMENS Arbeitsgruppe SW-Spezifikationsmethoden / Test- und Qualitätssysteme;
INTAKT, Interaktiver Arbeitsplatz zur Qualitätskontrolle von Software,
Funktionskatalog München, Oktober 1980

[ST 83] Schrupp,W.; Tamme,J.;
Spezifikation und Implementierung des Aufbereitungsteils von INTAKT,
Universität Bonn, FB Informatik, Diplom-Arbeit 1982

[THO 83] Thomas,C.;
Das Rewrite-Rule-Labor,
Universität Bonn, FB Informatik, Diplom-Arbeit,
(Stand 1983)