

8. Formale Spezifikation

Ziele:

Einordnung von Spezifikationen

Ansatz nach Parnas

Algebraische Spezifikationen

PROGRES-Spezifikation für Werkzeugbau

Einordnung/Klassifizierung von Spezifikationen

- Spezifikationsbegriff mehrdeutig
 1. Anforderungsspezifikation (vgl. Kap. 5)
im Sinne der Festl. Außenverhalten, Einbettung
 2. Entwurfsspezifikation (vgl. Kap. 6)
im Sinne Festlegung Bauplan
 3. Festl. eines Kernteils d. Systems (vgl. Prototyping)
zur Klärung
als Vorstufe zur Realisierung
als "Ableitung" des Codes der Realisierung
- ...
- formale Aufschreibung
- Pseudocode formalisieren

- Klassifikationsdimensionen
 - Niveau: Anforderungen, Bauplan, Detailrealisierung
 - Granularität: Innenleben Baustein, Baustein, ..., Gesamtsystem (je nach Sicht)
 - Vollständigkeit: Sicht, ..., vollst. Beschreibung
 - Formalisierung: semiformal SA, (E)ER, UML
formal
 - Ansätze: deklarativ, operational
 - zugr. Kalkül: Logik, Petrinetz, Graphgrammatiken etc.

- Spezifikation und Implementierung
 - manche Spezifikationen sind ausführbar
Vorabprüfung, Generierung, ist bereits Realisierung
 - Umsetzung einer Spezifikation abhängig von Niveau, Abstand, Hilfsmitteln

- im folgenden besprechen wir Spezifikationen
 - a) im Sinne einer Semantikfestlegung von Bausteinen eines Softwaresystems als zusätzliche Festlegung zu 2. von oben: Parnas, alg. Spezifikation
 - b) vgl. PROGRES zur Werkzeugespezifikation interner Datenstrukturen und deren Veränderungen bei Werkzeugaktivierungen im Sinne von 3. von oben: PROGRES

Spezifikation nach PARNAS

- gestützt auf D-Module (Information Hiding)
nicht alle Module sind D-Module!
- Modulspezifikation semiformal (formal (nichtalg.), textuell)
Gesamtsystem umgangssprachlich?
- Spezifikation nur der Zugriffsoperationen
(zustandsändernde, wertliefernde), nicht der internen
Datenstrukturen mit folgenden Angaben:
 - Wertebereich bei wertabliefernden Funktionen
 - Anfangswert vor Aktivierung einer zustandsändernden
Zugriffsoperation
 - Typen u. Namen der Parameter
 - Effekt (keiner operationelle Semantik (Überspez. !)
"Postconditions")

- Bsp.: Spez. nach Parnas

module stack (max):

function PUSH (a)

parameters: integer a

effect: if 'DEPTH' = max

then call ERROR1

else (TOP = a, DEPTH = 'DEPTH' + 1)

function POP

parameters: none

effect: if 'DEPTH' = 0

then call ERROR2

else DEPTH = 'DEPTH' - 1

function TOP

possible values: integer

initial values: undefined

parameters: none

effect: if 'DEPTH' = 0

then call ERROR3

function DEPTH

possible values: integer

initial value: 0

parameters: none

effect: none

Die Aufruffolge "PUSH(a); POP" hat keinen Effekt,
wenn kein Fehleraufruf auftritt.

end module stack.

Vorteile:

- Datenstrukturierung nicht festgelegt (Adaptabilität)
- Effektspezifikation nichtalgorithmisch $\Rightarrow$ keine Implementierungsfestlegung

Nachteile:

- Zusammenhänge zwischen Operationen nicht ausdrückbar (Text hierfür!)
- Vollständigkeit / Widerspruchsfreiheit schwer feststellbar
- nicht leicht verständlich, da Zusammenhänge der Bausteine ungeklärt

Algebraische Spezifikation (Zilles, Guttag u.a.)

- Übersicht
 - gestützt auf D-Module
 - formal
 - Fkt: aufbauend, zustandsverändernd, traversierend, informationsliefernd
 - syntaktischer Teil / semantischer Teil
Typen. d. Par. Axiome
d. Funktionen etc.
 - Obj. d. Typs BINTREE ergibt sich durch Sequenz
zustandsver. Funktionen (emptytree, make, left, right)
 - Axiome hier: Wirkung nichtaufbauender Funktionen
 - Funktionen hier alle seiteneffektfrei
 - auf rechter Seite von Axiomen:
Literale, Variablen, Junktoren, Ausdrücke, Fallauswahl,
Rekursion
 - Verständlichkeit und suggestive Bezeichner

- algebraische Spezifikation Binärbaum:

type BTREE

functions emptytree : $\emptyset$ $\rightarrow$ BTREE
 make : BTREE $\times$ CHAR $\times$ BTREE $\rightarrow$ BTREE
 is empty : BTREE $\rightarrow$ BOOL
 left : BTREE $\rightarrow$ BTREE
 right : BTREE $\rightarrow$ BTREE
 data : BTREE $\rightarrow$ CHAR
 is in : BTREE $\times$ CHAR $\rightarrow$ BOOL

axioms for l, r $\in$ BTREE; c, d $\in$ CHAR let

is empty (emptytree) = true
 is empty (make (l, c, r)) = false
 left (emptytree) = emptytree
 left (make (l, c, r)) = l
 right (emptytree) = emptytree
 right (make (l, c, r)) = r
 data (emptytree) = undefined
 data (make (l, c, r)) = c
 is in (emptytree, d) = false
 is in (make (l, c, r), d) =
 if c = d then true
 else is in (l, d) or is in (r, d)

end type BTREE

- Zur Bedeutung der Wahl sugg. Bezeichner

type MYSTERY

functions delete : $\emptyset$ $\rightarrow$ MYSTERY
 follow : MYSTERY $\times$ CHAR $\rightarrow$ MYSTERY
 say : MYSTERY $\rightarrow$ MYSTERY
 make : MYSTERY $\rightarrow$ CHAR
 is cold : MYSTERY $\rightarrow$ BOOL
 destroy: MYSTERY $\times$ MYSTERY $\rightarrow$ MYSTERY

axioms for $m, n \in \text{MYSTERY}; c \in \text{CHAR}$ let

is cold (delete) = true
 is cold (follow (m, c)) = false
 say (delete) = delete
 say (follow (m, c)) =
 if is cold (m) then delete
 else follow (say (m), c)
 make (delete) = undefined
 make (follow (m, c)) =
 if is cold (m) then c
 else make (m)
 destroy (m, delete) = m
 destroy (m, follow (n, c)) =
 follow (destroy (m, n), c)

end type MYSTERY

- Entschlüsselung des Typ MYSTERY

type QUEUE

functions emptyqueue : $\emptyset$ $\rightarrow$ QUEUE
 enqueue : QUEUE $\times$ CHAR $\rightarrow$ QUEUE
 dequeue : QUEUE $\rightarrow$ QUEUE
 front : QUEUE $\rightarrow$ CHAR
 is empty : QUEUE $\rightarrow$ BOOL
 concat : QUEUE $\times$ QUEUE $\rightarrow$ QUEUE

axioms for $m, n \in \text{QUEUE}$; $c \in \text{CHAR}$ let

is empty (emptyqueue) = true
 is empty (enqueue (m, c)) = false
 dequeue (emptyqueue) = emptyqueue

dequeue (enqueue (m, c)) =
 if is empty (m) then emptyqueue
 else enqueue (dequeue (m), c)

front (emptyqueue) = undefined

front (enqueue (m, c)) =
 if is empty (m) then c
 else front (m)

concat (m, emptyqueue) = m

concat (m, enqueue (n, c)) =
 enqueue (concat (m, n), c)

end type QUEUE

- Bsp. Datentypgenerator
(parametrisierter aDt, generischer aDt)

type generator SET [ITEM]

<u>functions</u> emptyset :	$\emptyset$	$\rightarrow$ SET
add :	SET $\times$ ITEM	$\rightarrow$ SET
is in :	SET $\times$ ITEM	$\rightarrow$ BOOL
delete :	SET $\times$ ITEM	$\rightarrow$ SET

axioms for $s \in \text{SET}; i, j \in \text{ITEM}$ let

add (add (s, i), j)	=
if i = j <u>then</u> add (s,i)	
<u>else</u> add(add (s, j), i)	
is in (emptyset, j)	= false
is in (add (s, i), j)	=
<u>if</u> i = j <u>then</u> true	
<u>else</u> is in (s, j)	
delete (emptyset, j)	= emptyset
delete (add (s, i), j)	=
<u>if</u> i = j <u>then</u> s	
<u>else</u> add (delete (s, j), i)	

end type generator SET

- Versteckte (priv., nichtöff.) Funktionen

type generator FSTACK [ITEM]

functions mtstack : NAT $\rightarrow$ FSTACK
 hiddenpush:FSTACK $\times$ ITEM $\rightarrow$ FSTACK
 push : FSTACK $\times$ ITEM $\rightarrow$ FSTACK
 pop : FSTACK $\rightarrow$ FSTACK
 top : FSTACK $\rightarrow$ ITEM
 depth : FSTACK $\rightarrow$ NAT
 limit : FSTACK $\rightarrow$ NAT

axioms for $k \in \text{NAT}; i \in \text{ITEM}; s \in \text{FSTACK}$ let

pop (mtstack (k)) = error1
 pop (hiddenpush (s, i)) = s
 top (mtstack (k)) = undefined
 top (hiddenpush (s, i)) = i
 depth (mtstack (k)) = 0
 depth (hiddenpush (s, i)) = depth (s) + 1
 limit (mtstack (k)) = k
 limit (hiddenpush (s, i)) = limit (s)
 push (s, i) =
 if depth (s) < limit (s) then hiddenpush (s, i)
 else error2

end type generator FSTACK

- Einführung Fehlersituationen

type generator FSTACK [ITEM]

<u>functions</u> mtstack :	NAT	→ FSTACK
push :	FSTACK × ITEM	→ FSTACK
pop :	FSTACK	→ FSTACK
top :	FSTACK	→ ITEM
depth :	FSTACK	→ NAT
limit :	FSTACK	→ NAT

axioms for $k \in \text{NAT}$; $i \in \text{ITEM}$; $s \in \text{FSTACK}$ let

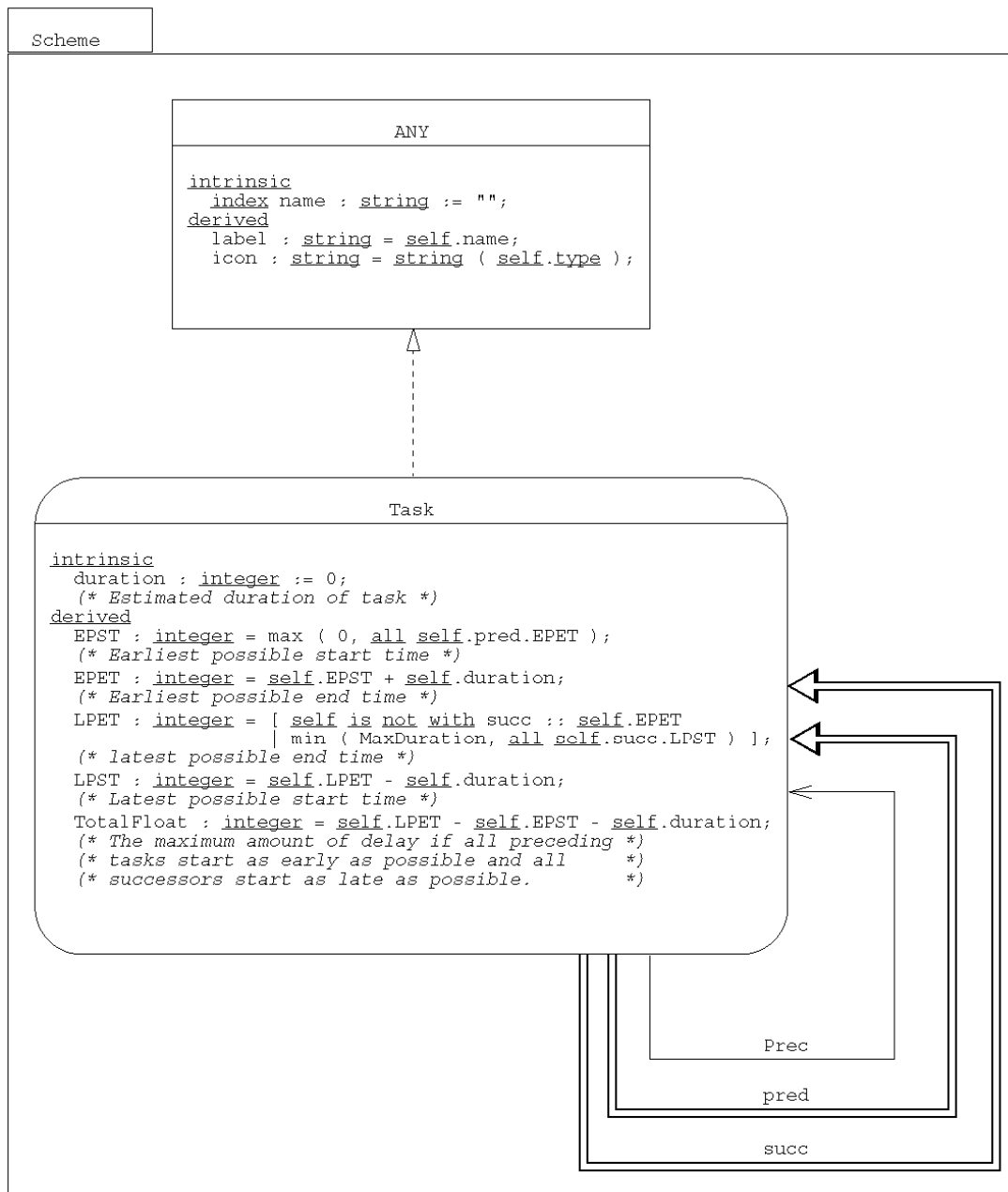
pop (push (s, i))	= s
top (push (s, i))	= i
depth (mtstack (k))	= 0
depth (push (s, i))	= depth (s) + 1
limit (mtstack (k))	= k
limit (push (s, i))	= limit (s)

restrictions

depth (s) ≥ limit (s)	⇒	push (s, i) = error2
depth (s) = 0	⇒	pop (s) = error1
depth (s) = 0	⇒	top (s) = undefined

end type generator FSTACK

- Vorteile:
 - formal
 - implementierungsunabhängig
 - erweiterbar
(weitere Funktionen, weitere Axiome)
 - parametrisierbare Datentypen
(gen. abstr. Datentypen)
- Nachteile:
 - Fehlerbehandlung
 - nur D-Module
 - versteckte Funktionen
endl. Datentypen
 - Spez. für Gesamtplan nötig:
hier beschränkt auf Einzelmodule
Zusammenhänge zwischen Bausteinen fehlen




```

spec SimplePNets

  section Scheme

    node_class ANY
    [...]
  end;

    node_type Task : ANY
    [...]
  end;

    edge_type Prec : Task -> Task;
    (* precedence relationship *)

    path succ : Task -> Task =
      -Prec->
    end;

    path pred : Task -> Task =
      <-Prec-
    end;

  end;

  section HelperFunctions

    function MaxDuration : -> integer =
      100
    end;

    function min : ( v1, v2 : integer) -> integer =
      [ v1 > v2 ;; v2
        v1 ]
    end;

    function max : ( v1, v2 : integer) -> integer =
      [ v1 < v2 ;; v2
        v1 ]
    end;

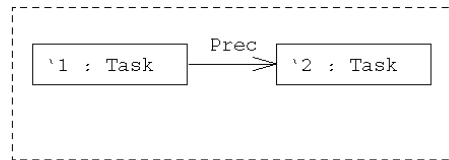
  end;

```

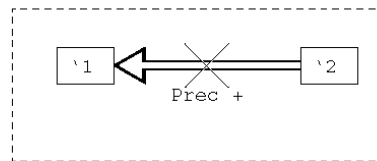
section TaskOperations

section EditOperations

constraint
for



ensure

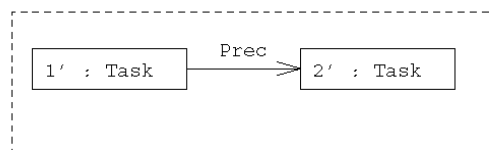


end;
(Ensure acyclic task nets *)*

production CreateDiagram(StartName, StopName : string) =



:=



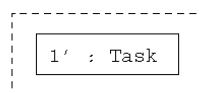
transfer 1'.name := StartName;
2'.name := StopName;

end;

production CreateTask(Name : string) =

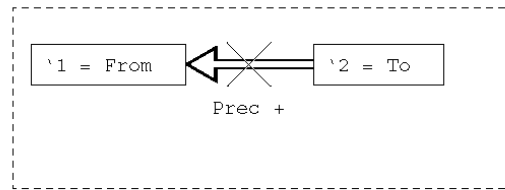


:=

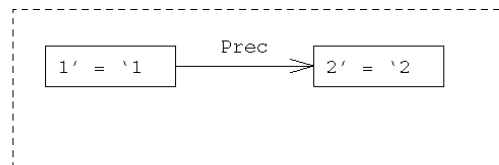


transfer 1'.name := Name;
end;

```
production CreatePrecedence( From, To : Task) =
```

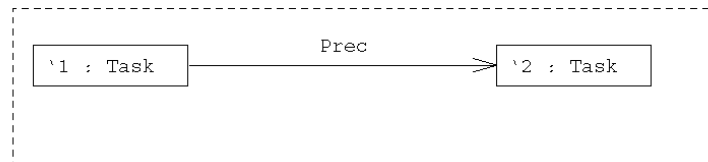


```
:=
```

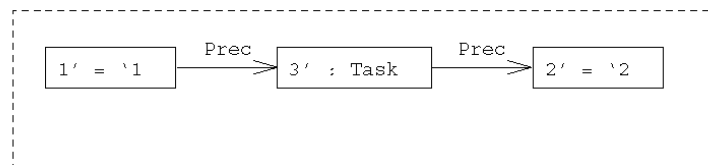


```
end;
```

```
production InsertTask( FromName, NewName, ToName : string) =
```

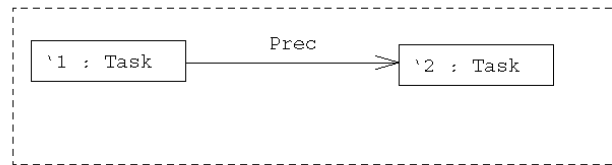


```
:=
```

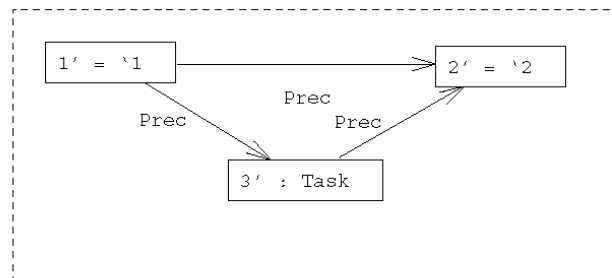


```
condition '1.name = FromName;  
         '2.name = ToName;  
transfer '3'.name := NewName;  
end;
```

```
production ParallelTask( FromName, NewName, ToName : string)
=
```

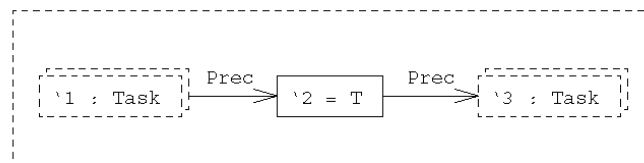


```
 ::=
```

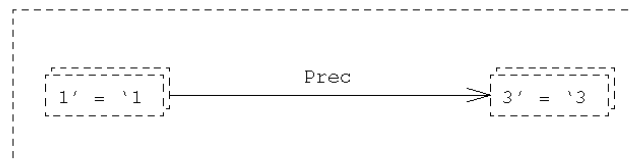


```
condition '1.name = FromName;
          '2.name = ToName;
transfer 3'.name := NewName;
end;
```

```
production DeleteTask( T : Task) =
```



```
 ::=
```



```
end;
```

Spezifikationsproblematik: Wertung

- bisher kennengelernt:

SA/EER/CTR	für Anforderungsspezifikationen
UML	(funkt. Teil)
Architektursprache(n)	für Architekturspezifikation
UML	für Grafiksicht Architektur
PROGRES	für Werkzeugbau
ET	für Fallunterscheidungshierarchien
Statecharts	für Systeme der Kategorie "endl. Automaten"

- in diesem Kapitel:

Parnas
Algebr. Spezifikation }
}

nur Datenabstraktion ist künstlich
formal: Ausbildungsstand u. Qualifikation d.
Programmierer
für konkrete Systeme sehr kompliziert
geben kaum Anleitung f. Entwurfsprozess
eher auf Modul- als Systemarchitekturebene

PROGRES

für abstr. Realisierung (Kernteil eines interaktiven
„Design“-Werkzeugs)