Spezifizieren mit Graphersetzungssystemen

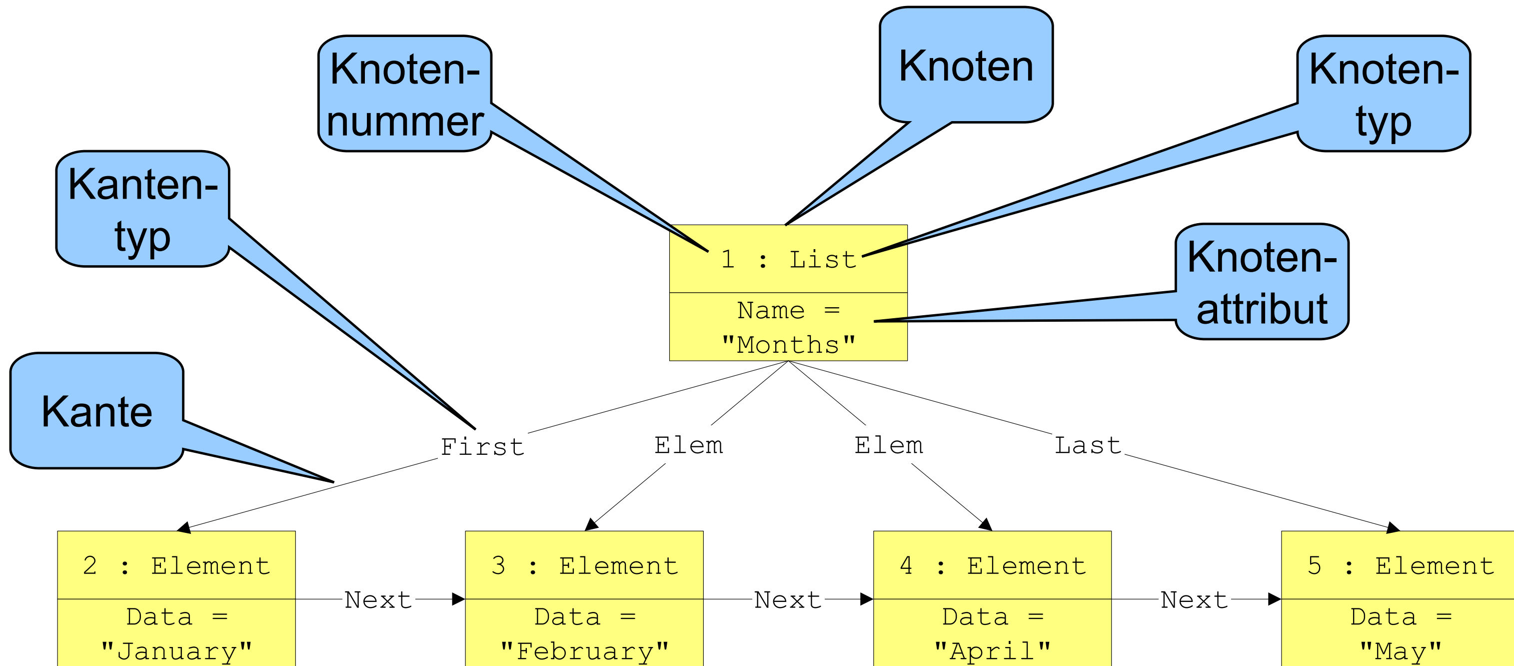
Bernhard Westfechtel

Lehrstuhl für Informatik III, RWTH Aachen

Charakterisierung

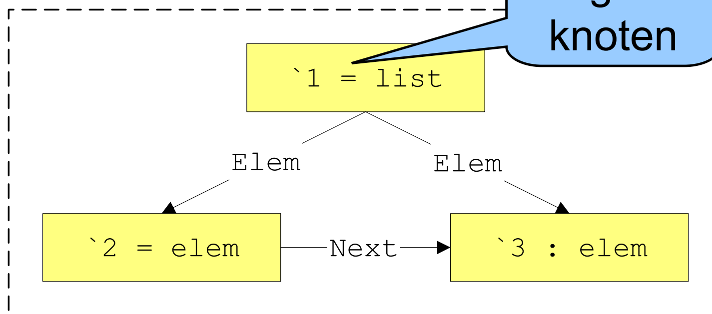
- Formale Spezifikation abstrakter Datentypen
- Modellorientierte Spezifikation
- Graphen als zugrundeliegendes Datenmodell
- Spezifikation lesender Operationen durch Graphtests, verändernder Operationen durch Graphersetzungsregeln
- Beweistechnik: Induktion
- Rapid Prototyping durch Generierung von Code aus der Spezifikation

Darstellung einer Liste als Graph

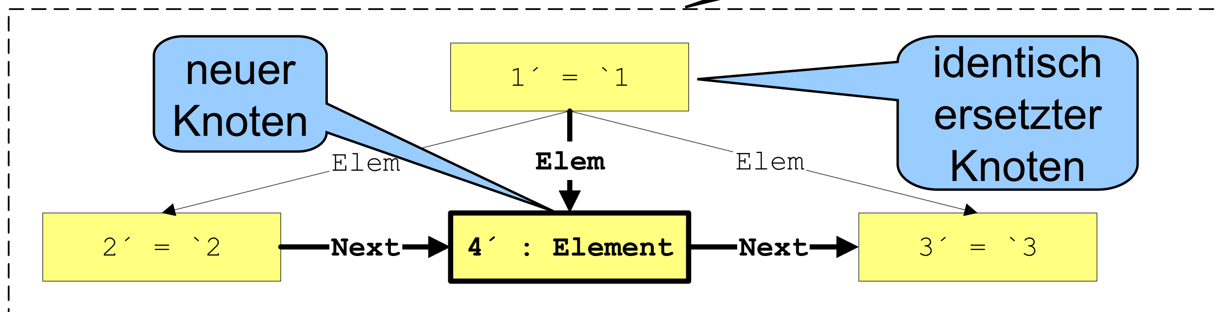


Beispiel für eine Graphersetzungsregel

```
production PostInsertElement
    (list : List; elem : Element; data : string;
    out new : Element) =
```

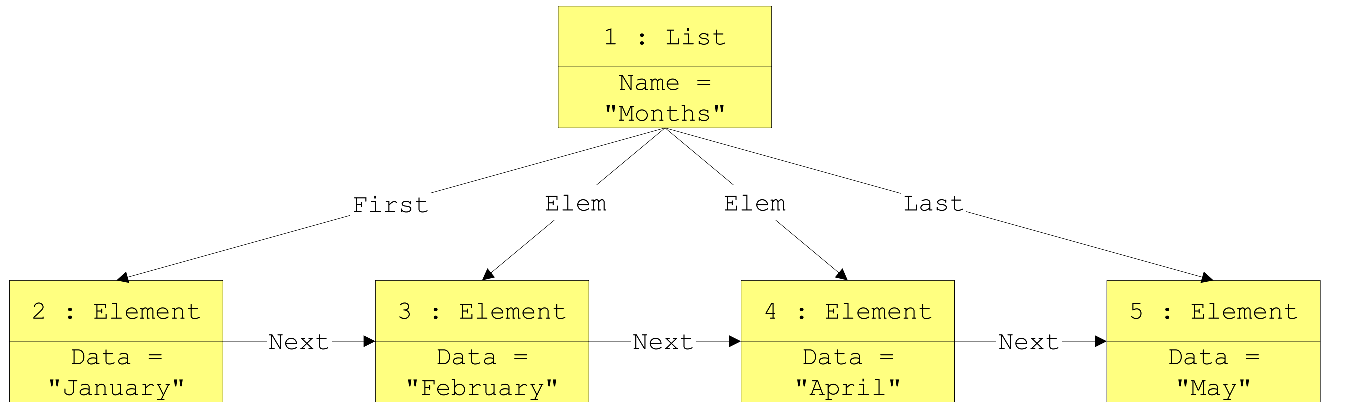



```
::=
```

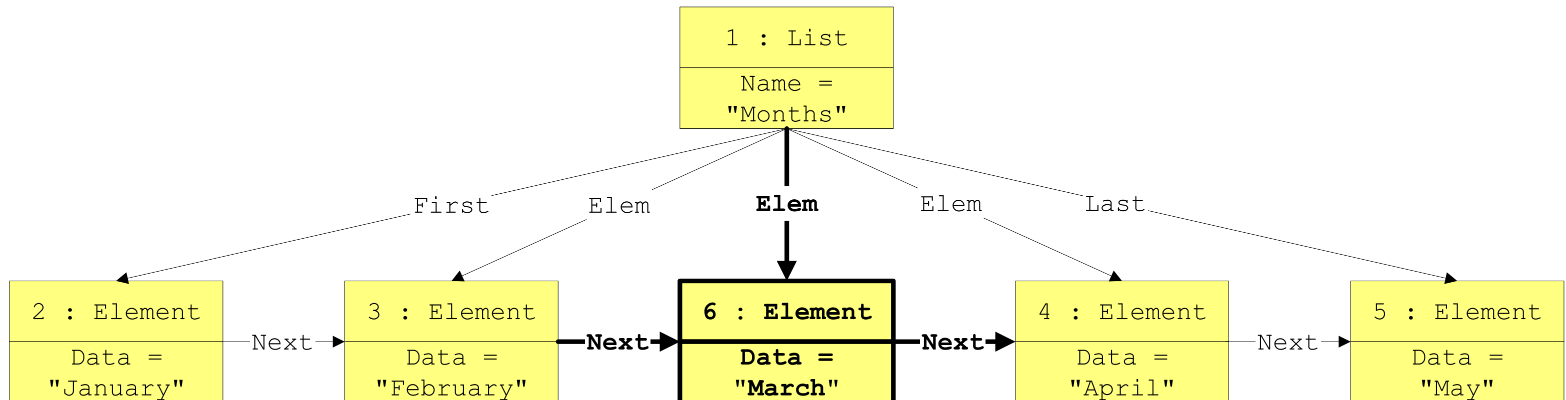



```
    attribute transfer 4´.Data := data;
    return new := 4´;
  end;
```

Anwendung der Graphersetzungsregel



PostInsertElement(1, 3, "March", out 6)



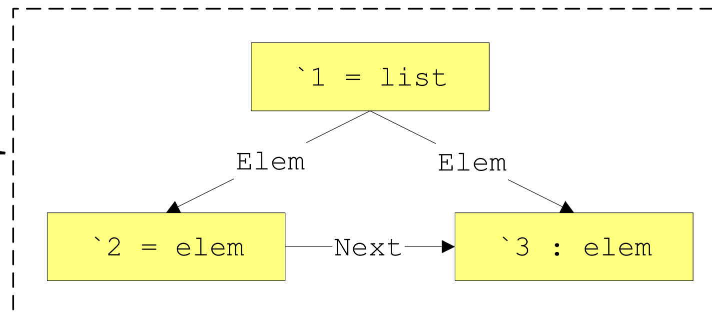
Beispiel für einen Graphtest

Name

Parameter

```
test GetNextElement
   (list : List; elem : Element; out next : Element) =
```

Graph




```
    return next := ´3;
end;
```

Rück-
gabeteil

Graphen

Definition: Gerichteter, markierter Graph

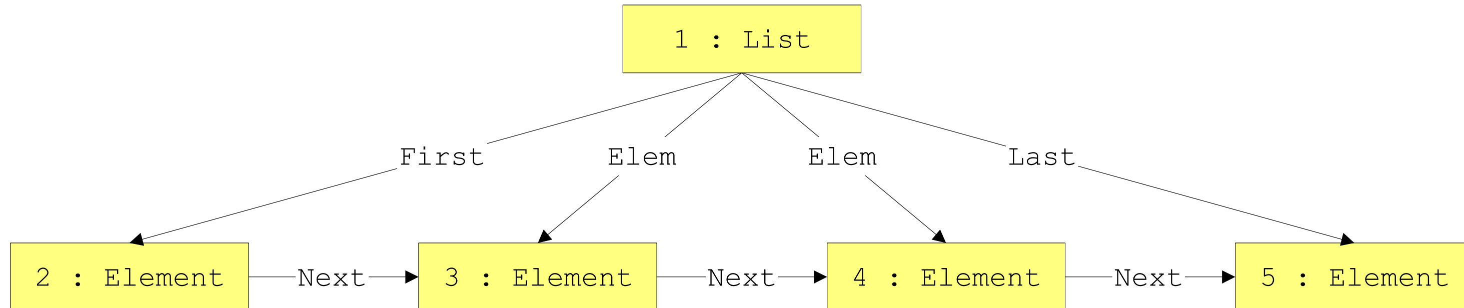
$G = (V, E, l)$ ist ein gerichteter Graph über Markierungsmengen L_V (labels for vertices) und L_E (labels for edges) $\Leftrightarrow$

» V ist eine Menge von Knoten(-bezeichnern).
» $E \subseteq V \times L_E \times V$ ist eine Menge markierter Kanten.
» $l : V \rightarrow L_V$ ist die Markierungsfunktion für Knoten.

Bemerkungen:

- Nur Knoten besitzen Bezeichner, nicht aber Kanten.
- Deshalb gibt es keine parallelen Kanten mit derselben Markierung.
- Kanten sind zweistellige Relationen.
- Knoten und Kanten sind getypt (markiert).
- Weder Knoten noch Kanten sind bislang attributiert.

Beispiel für einen gerichteten Graphen



$G = (V, E, l)$ mit:

- $V = \{1, 2, 3, 4, 5\}$
- E = { (1, First, 2), (1, Elem, 3), (1, Elem, 4), (1, Last, 5), (2, Next, 3), (3, Next, 4), (4, Next, 5)}
- l = { (1, List), (2, Element), (3, Element), (4, Element), (5, Element)}

Graphersetzungsregeln

Definition: Graphersetzungsregel

Eine Graphersetzungsregel ist ein Tripel $r = (L, K, R)$ mit:

» L ist ein Graph, die linke Regelseite von r.
» R ist ein Graph, die rechte Regelseite von r.
» $K = L \cap R$ ist der Klebegraph.

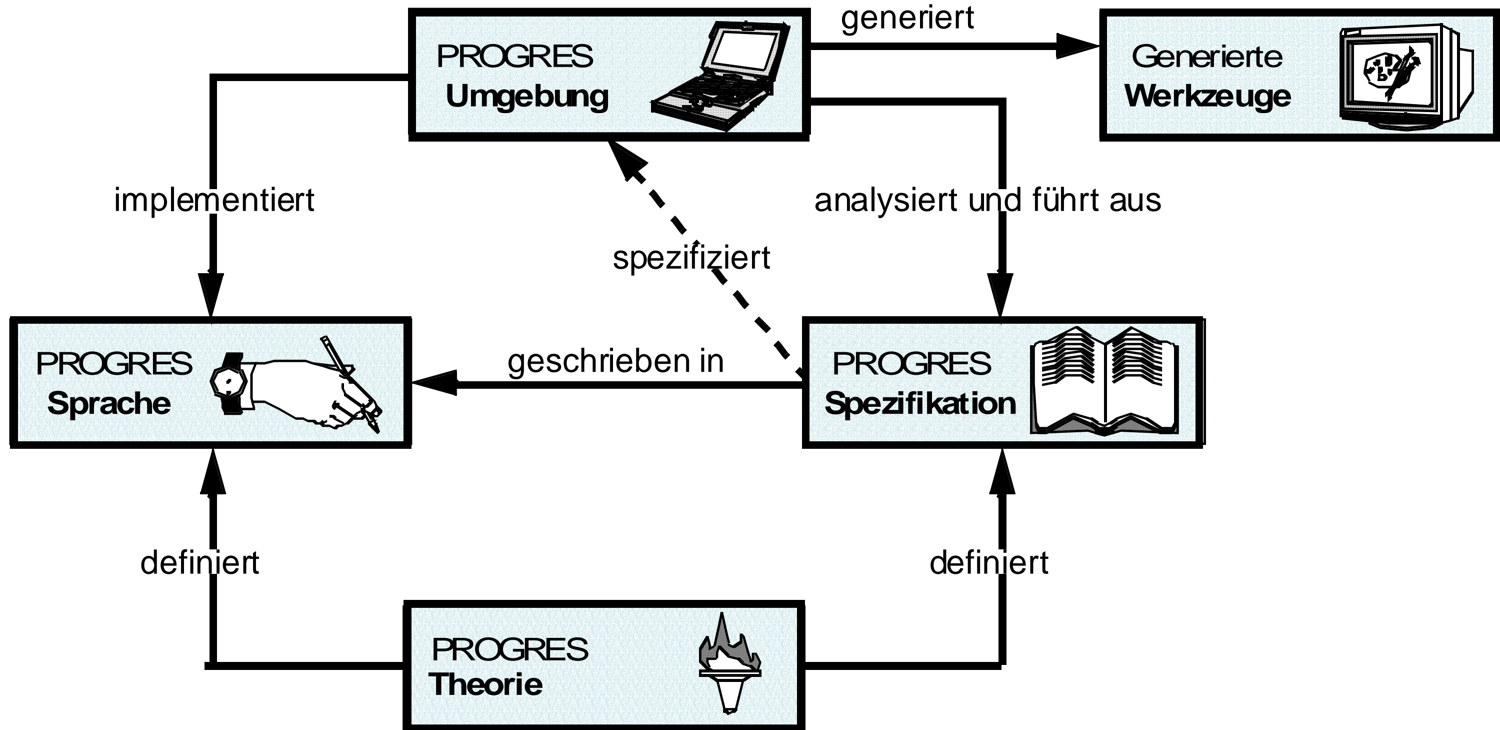
Bemerkungen

- Regel löscht alles, was Element von L aber nicht von K ist.
- Regel erzeugt alles, was Element von R aber nicht von K ist.
- Regel bewahrt alles, was zu K gehört.
- K heißt Klebegraph, weil er zur Verbindung der neuen Knoten aus R dient.

Was ist PROGRES?

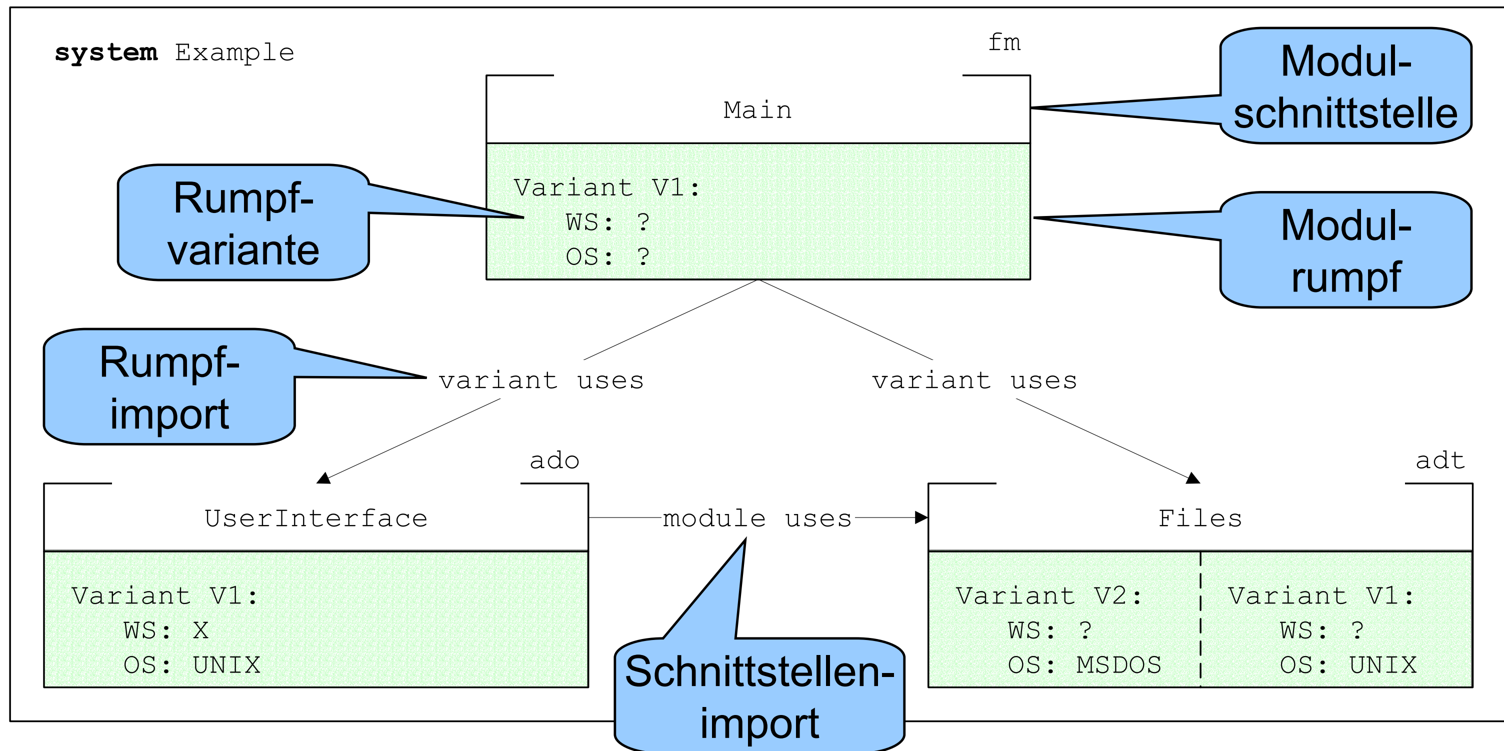
- **PROGRES** = **PRO**grammierte **GR**aph-**E**rsetzungs-**S**ysteme
- Multiparadigmatische **Spezifikationssprache**, basierend auf Graphersetzungen
  - Objektorientierte Modellierung von Graphschemata
  - Deklarative Definition und inkrementelle Berechnung abgeleiteter Attribute
  - Regelorientierte und visuelle Beschreibung von Graphtests und Graphtransformationen
  - Imperative und nichtdeterministische Programmierung
- **Entwicklungsumgebung** zur Erstellung von Spezifikationen
  - Syntaxgestützter Editor
  - Statische Analysen
  - Interpreter
  - Codegenerierung

Komponenten von PROGRES

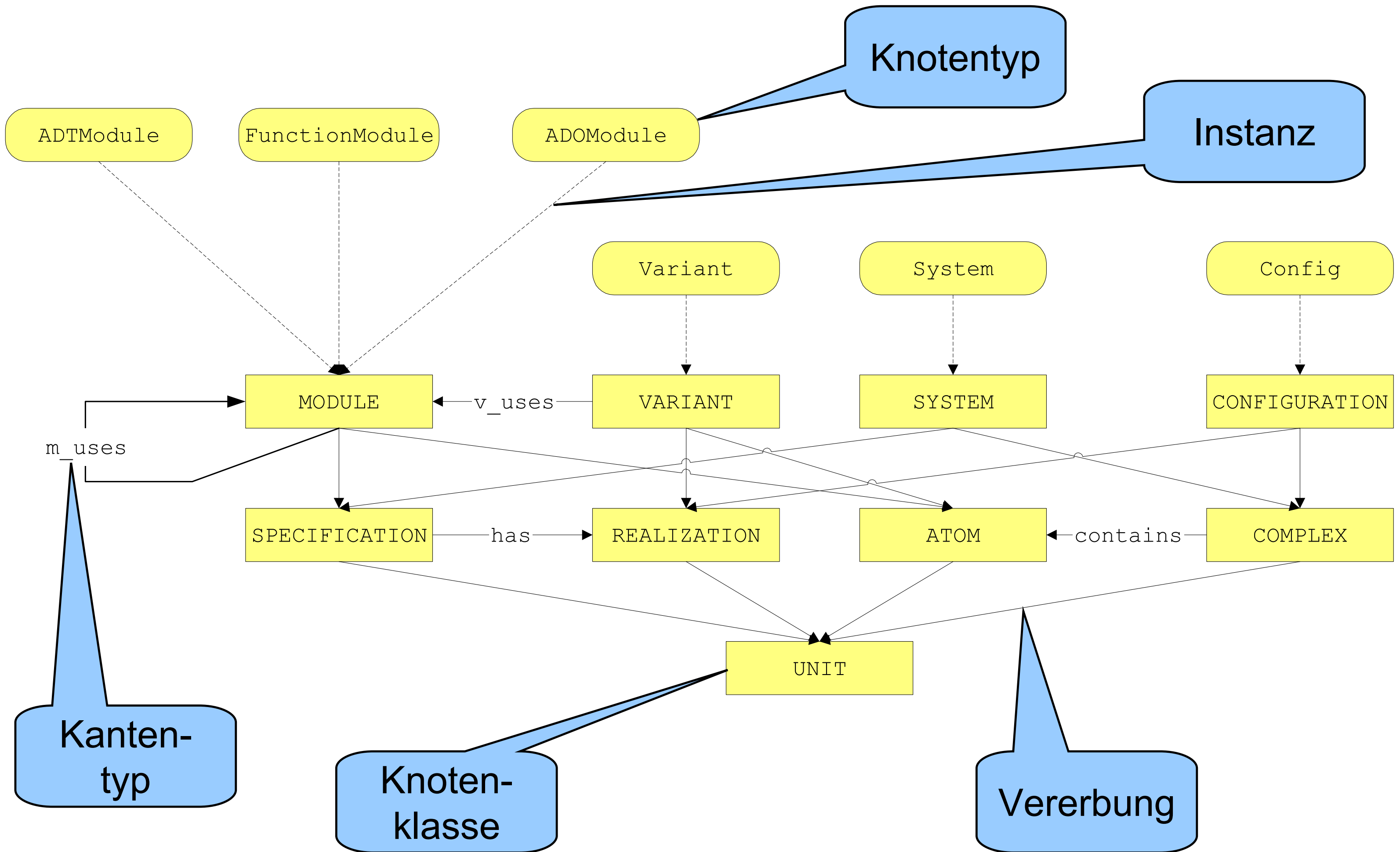


Anwendungsbeispiel

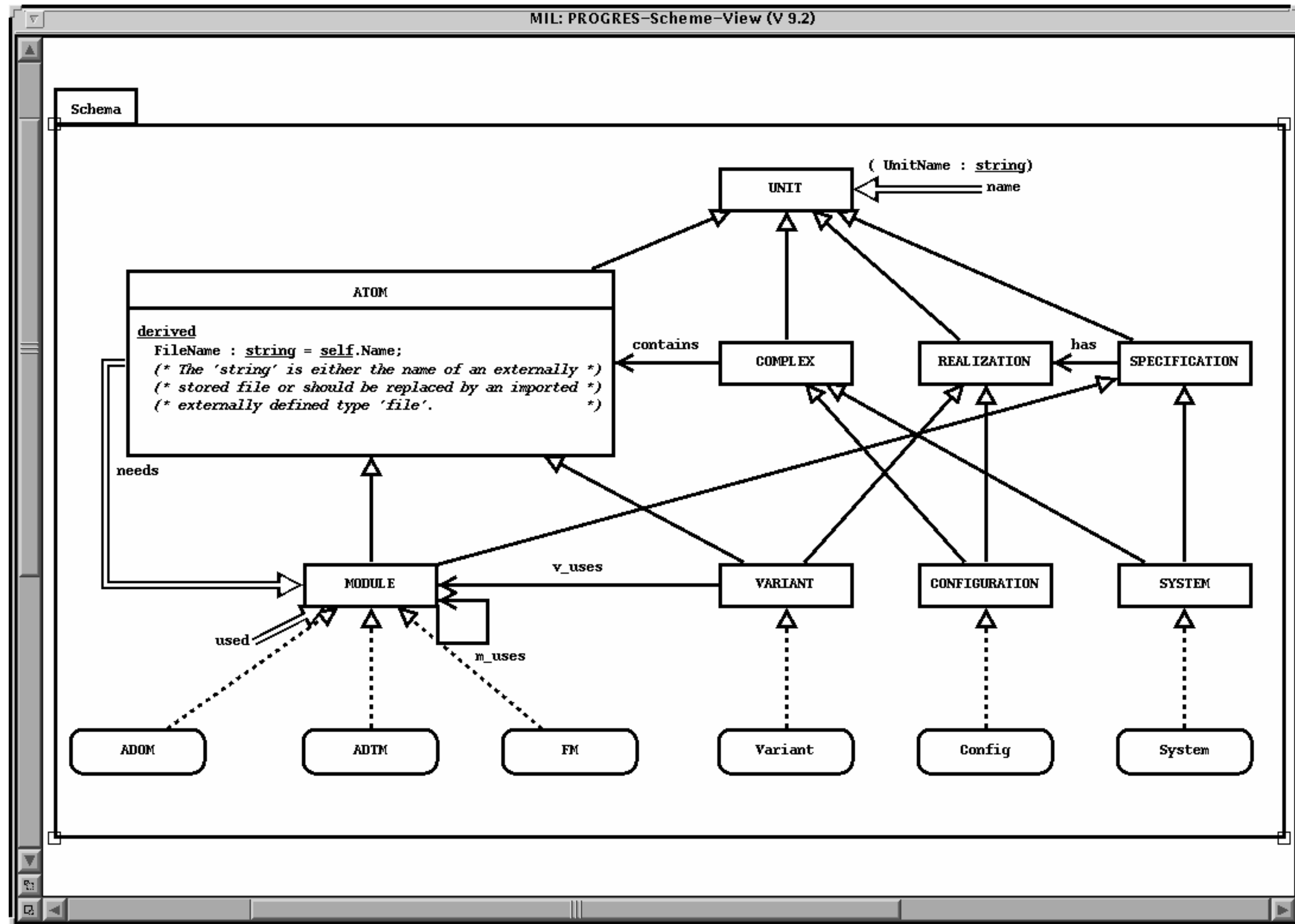
Werkzeuge für das Programmieren im Großen



Graphische Definition eines Graphschemas



PROGRES-Schemaeditor



Definition von Attributen

- Ein Graphschema kann sowohl graphisch als auch textuell eingegeben werden.
- Die Attribute und Attributberechnungsvorschriften können nur in der textuellen Sicht definiert werden.
- Intrinsische Attribute erhalten ihren Wert durch explizite Zuweisungen und können mit einer Konstanten initialisiert werden.
- Abgeleitete Attribute errechnen sich aus den Attributwerten benachbarter Knoten gemäß einer angegebenen (gerichteten) Gleichung.
- Benachbarte Knoten sind alle die Knoten, die durch einen sogenannten Pfadausdruck erreicht werden, z.B.:
  - `self` : liefert den aktuellen Knoten
  - `-e->` bzw. `<-e-` : Traversieren von Kanten des Typs `e` (vor- bzw. rückwärts)
  - Konkatenation : `p1 & p2`

Textuelle Definition eines Graphschemas (1)

```
node class UNIT
   derived
      Size : integer = 0;
   intrinsic
      Name : string := "";
end;     (* Root of class hierarchy. *)

node class SPECIFICATION is a UNIT
   redef derived
      Size = max( 0, all self.-has->.Size );
        (* The 'Size' of a specification is the maximum of     *)
        (* the size of all its realizations, instead of        *)
        (* being the sum (which would reasonable, too).        *)
end;    (* A 'SPECIFICATION' is either the complete design of *)
        (* a software system or a design of one its parts.     *)

node class REALIZATION is a UNIT
   intrinsic
      Props : string [0:n];  (* Set of guaranteed properties *)
end;     (* A 'REALIZATION' is an implementation of a 'SPEC.' *)
         (* which fulfills a given set of properties like      *)
         (* needed hardware platforms, operating system    *)
```

Textuelle Definition eines Graphschemas (2)

```
edge type has : SPECIFICATION [1:1] -> REALIZATION [0:n];
        (* A 'SPECIFICATION' 'has' an arbitrary number of    *)
        (* 'REALIZATIONS', but a 'REALIZATION' belongs to a  *)
        (* uniquely defined 'SPECIFICATION'.             *)

node class COMPLEX is a UNIT
   redef derived
      Size = addSize( 0, all self.-contains-> );
       (* The 'Size' of a complex unit is the sum of *)
       (* the 'Size' of all its children.            *)
end;

node class ATOM is a UNIT
   intrinsic
      File : file;    (* Pointer to externally stored file. *)
   redef derived
      Size = size( self.File );
             (* 'Size' is the length of the attached File. *)
end;

edge type contains : COMPLEX [1:1] -> ATOM [0:n];
        (* A 'COMPLEX' may contain 0 to n 'ATOM' nodes. *)
        (* Conversely, an 'ATOM' node is contained in exactly
        (* one'COMPLEX'. *)
```

Kantentyp

Kardinalität

Pfadausdrücke und Restriktionen

- Ein **Pfadausdruck** ist
  - » zum einen eine abgeleitete Relation zwischen Knoten (Kanten sind intrinsische Relationen):
    v_1 =p=> v_2 ⇔ Es führt ein Pfad p von v_1 zu v_2
  - » zum anderen eine Funktion auf Knotenmengen:
    $p(V) = \{ v_2 \mid \exists\, v_1 \in V : v_1 =p\Rightarrow v_2 \}$
- Eine **Restriktion** ist ein "einstelliger Pfadausdruck", d.h. es werden aus einer Menge von Knoten diejenigen Knoten ermittelt, die eine bestimmte Bedingung erfüllen.
- Sowohl Pfadausdrücke als auch Restriktionen lassen sich notieren:
  - » textuell
  - » graphisch

Beispiele für textuelle Pfadausdrücke

```
path needs : ATOM [0:n] -> MODULE [0:n] =
     (* The path 'needs' connects any variant or module to its imports. *)
      ( instance of MODULE  & =moduleNeeds=>  )
   or ( instance of VARIANT & =variantNeeds=> )
end;

path moduleNeeds : MODULE [0:n] -> MODULE [0:n] = -m_uses-> end;

path variantNeeds : VARIANT [0:n] -> MODULE [0:n] =
  -v_uses-> or ( <-has- & instance of MODULE & -m_uses-> )
end;

path dependsOn : MODULE [0:n] -> MODULE [0:n] =
   (* connects module to its interface imports & imports of its variants. *)
   ( self or -has-> ) & =needs=>+
end;
```

Beispiel für einen graphischen Pfadausdruck

Pfad

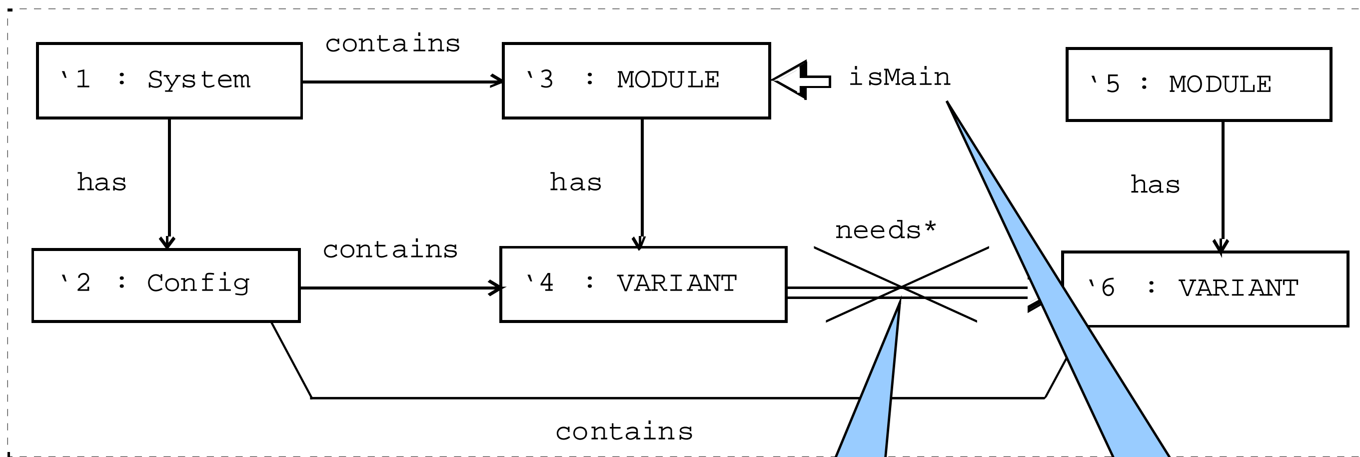
Quellklasse

Zielklasse

Quellknoten

Zielknoten

```
path UselessVariant : Config [0:n] -> VARIANT [0:n] = from '2 to '6 in
```



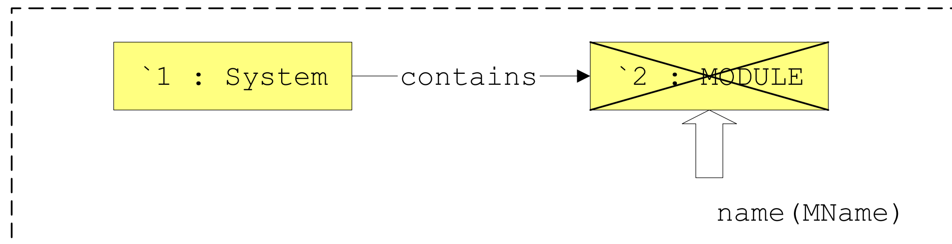

```
end;

(* searches for all variants which
are contained in a configuration
but are not reachable from the
top-level module *)
```

(Negativer) Pfad

Restriktion

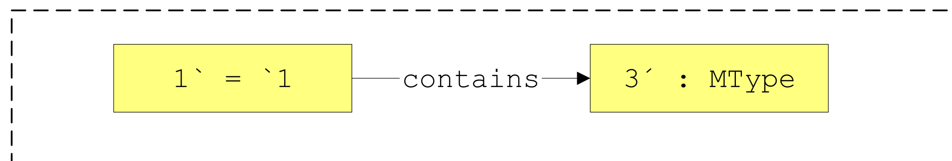
Beispiel für Parameter, negative Knoten, Attributtransfer etc.

```
production CreateModule(MName : string; InterfaceDescription : file;
                        MType : type in MODULE; out NewM : MODULE) =
```



(* Creates a module with name ´MName´if the ´System´ does not already contain a module with this name. The ´Mtype´ parameter is a ´FunctionModule´ or ´ADTModule´ or ´ADOModule´. *)

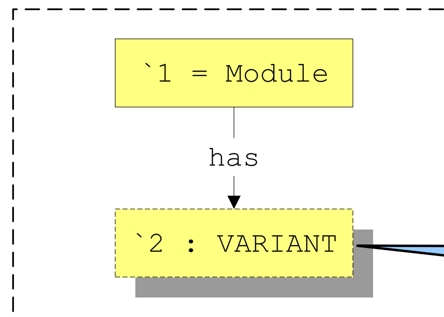
::=




```
    condition `2.Name = MName; (* Conditions only for normal nodes *)
    transfer 3´.Name := MName;
             3´.File := InterfaceDescription; (* External file handle. *)
    return   NewM := 3´;
end;

restriction name(UName : string) : UNIT = valid (self.Name = UName) end;
(* The restriction is valid if the current ´UNIT´ is named ´UName´. *)
```

Beispiel für optionalen Mengenknoten

```
production DeleteModule(Module : MODULE) =
```



Optionale Menge

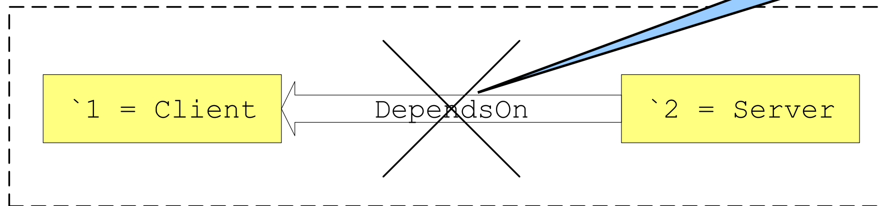
```
::=
```

```
end;
```

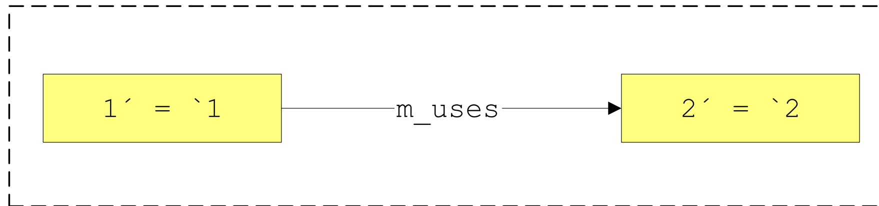
Beispiel für einen negativen Pfad

Negativer Pfad

production CreateMUse(Client, Server : MODULE) =



::=

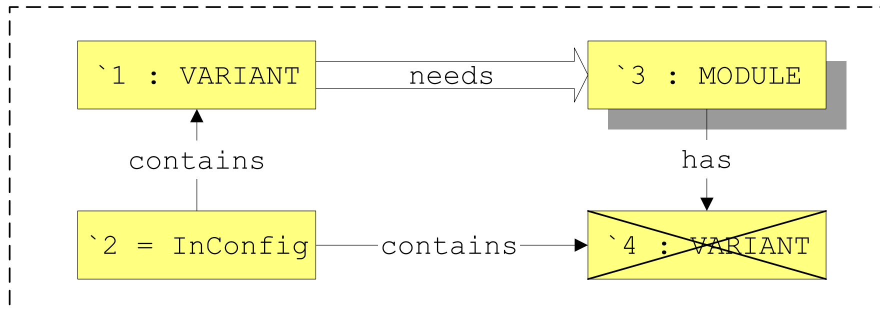


end;

(* Creates a new import from ´Client´ to ´Server´. *)

Beispiel für einen Graphtest

```
test UnresolvedImport(InConfig : Config; out MSet : MODULE [1:n]) =
```




```
   return MSet := `3;
end;
(* Returns all modules which are needed by some variant already
selected but for which no variant is included yet in the
configuration. *)
```

PROGRES Regeleditor

MIL: PROGRES-View-1 (V 9.2)

```
production ResolveImport( C : CONFIGURATION ; ReqProps : string [0:n] ;
                          out NewProps : string [0:n])
=
```

'4 : VARIANT
needs
'3 : MODULE
contains
has
'1 = C
contains
'2 : VARIANT
has
contains
'5 : VARIANT

```
::=
```

4' = '4
3' = '3
contains
has
1' = '1
contains
2' = '2

```
  condition '2.Props 'are_in' ReqProps;
  return NewProps := merge ( ReqProps, 2'.Props );
end;
(* Selects a module which is used by a 'Variant' in    *)
```

Executable Commands

EDIT
ANALYZE
BROWSE
CONSTRAINT
DISPLAY
INTERPRET
LAYOUT
MISC
UN/REDO
VERSION
HELP
QUIT

Generic
ChangeCard
RestrictCond (Er)
NotPathCond (E~)
NotEdgeDecl (En)
PathCond (Ep)
EdgeDecl (Ee)
NotNodeDecl (E-)
OptSetDecl (E*)
OblSetDecl (E+)
OptNodeDecl (E0)
OblNodeDecl (E1)

Cmd (abbr):

Kontrollstrukturen: Motivation und Eigenschaften

- Zusammensetzen von Graphtests zu komplexen **Queries** (die den Wirtsgraphen unverändert lassen)
- Zusammensetzen von Graphersetzungsregeln (und Graphtests) zu komplexen **Transaktionen**
- Für Transaktionen gelten (wie auch schon für Graphersetzungsregeln) die **ACID**-Eigenschaften von Datenbanktransaktionen:
  - » **A**tomic: Entweder vollständige erfolgreiche Durchführung oder keine Veränderung des Wirtsgraphen
  - » **C**onsistent: Konsistenzerhaltende Transformation
  - » **I**solated: Isolation im Mehrbenutzerbetrieb
  - » **D**uration: Dauerhaftigkeit
- Zusätzliche Eigenschaft: **Nichtdeterminismus**
- Implementierung führt nach einer Fehlentscheidung Backtracking aus

Übersicht über Kontrollstrukturen

Kontrollstruktur	**Bedeutung**
`p & q`	Sequenz
`p and q`	`p` und `q` in beliebiger Reihenfolge
`p or q`	Nichtdeterministische Auswahl
`choose p1 else p2 ... end`	Erst `p1`, dann `p2` ... versuchen
`loop p end`	Schleife (`p` so lange wie möglich)
`for all n do p end`	`p` für alle Knoten `n` ausführen
`use v : ... do p end`	Block mit Deklaration von Variablen

Zusammenfassung

Vorteile des Spezifizierens mit Graphersetzungsregeln

- Graphen sind ein geeignetes Datenmodell für eine große Klasse von Anwendungen.
- Mit Graphen lassen sich auch komplexe Datenstrukturen mit einer Vielzahl von Konsistenzbedingungen beschreiben.
- Durch Graphersetzungsregeln lassen sich auch komplexe Graphtransformationen deklarativ beschreiben.
- Visuelle Programmierung von Graphtransformationen ist anschaulich.
- Dennoch ist die Spezifikation operational, und es kann aus ihr Code generiert werden (Rapid Prototyping).

Nachteile des Spezifizierens mit Graphersetzungsregeln

- Die Allgemeinheit ist eingeschränkt: Festlegung auf ein Graph-Datenmodell.
- Für einfache Datentypen sind Graphen und Graphersetzungen ein Overkill.
- Mögliche Effizienzprobleme (Teilgraphensuche ist NP-vollständig).
- Im Falle von PROGRES:
  - » Sehr ausdrucksstarke, aber auch sehr komplexe Sprache.
  - » Spezifizieren im Großen noch nicht voll ausgearbeitet.