Specification with Graph Rewriting Systems

Bernhard Westfechtel

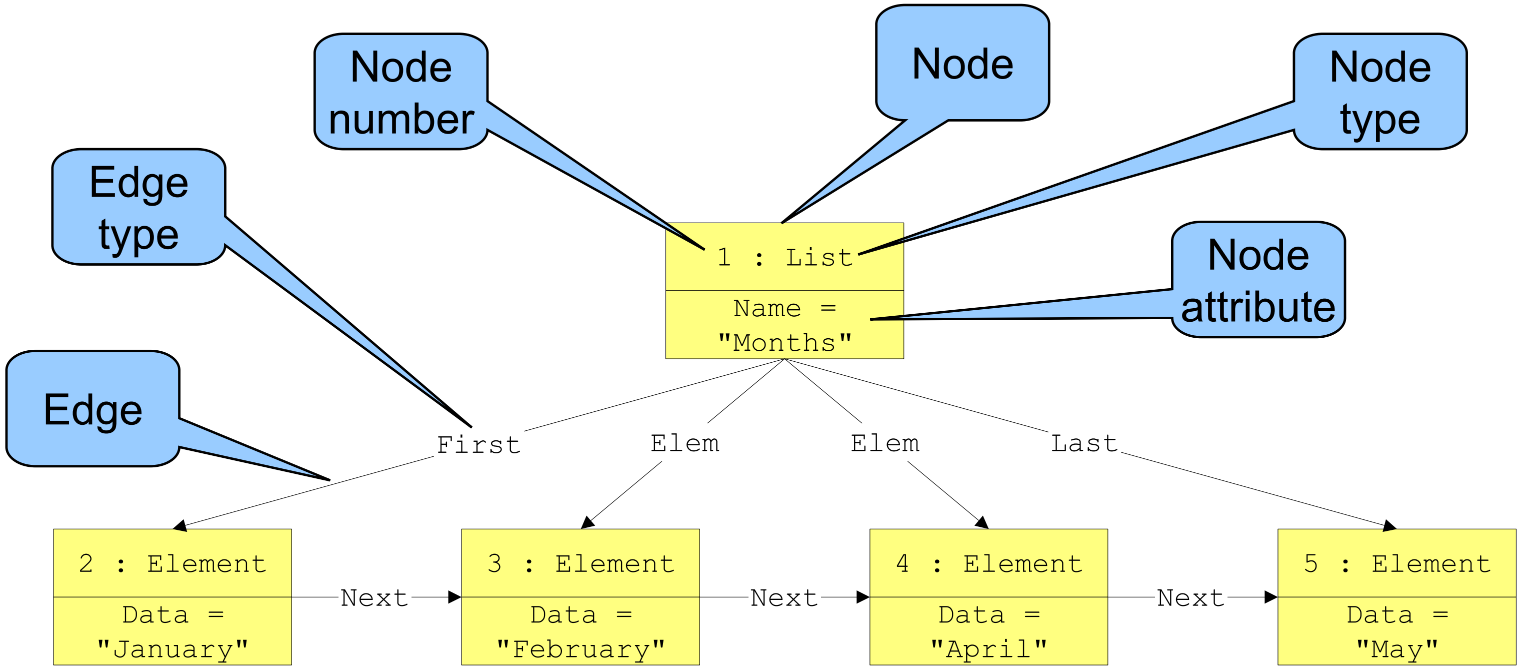
Lehrstuhl für Informatik III, RWTH Aachen

Characterization

- Formal specification of abstract data types
- Model-oriented specification
- Graphs as underlying data model
- Specification of read operations by graph tests,
  specification of write operations by graph rewrite rules
- Proof technique: induction
- Rapid Prototyping by generating code from the specification

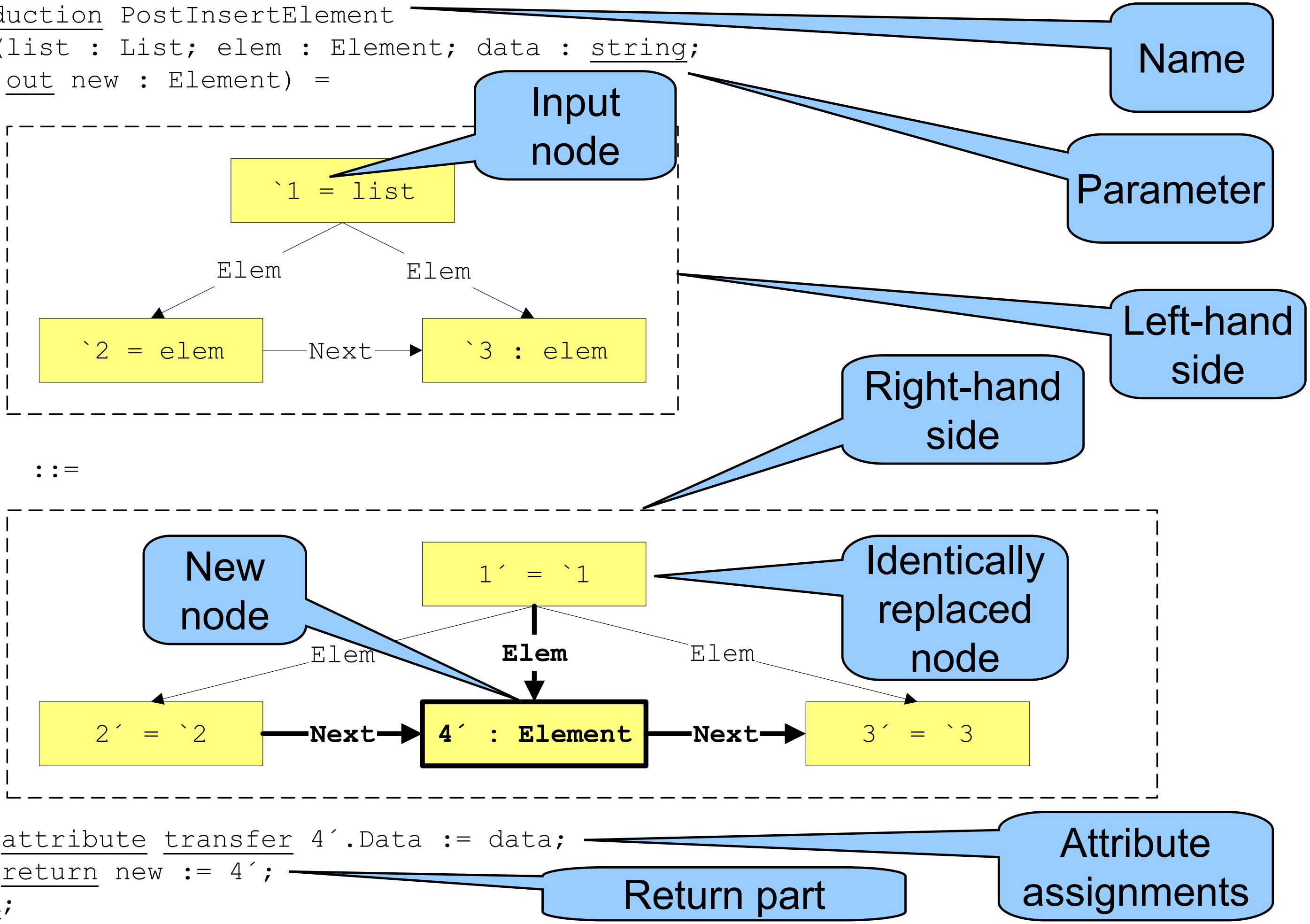
Introductory Example

Representation of a list as a graph



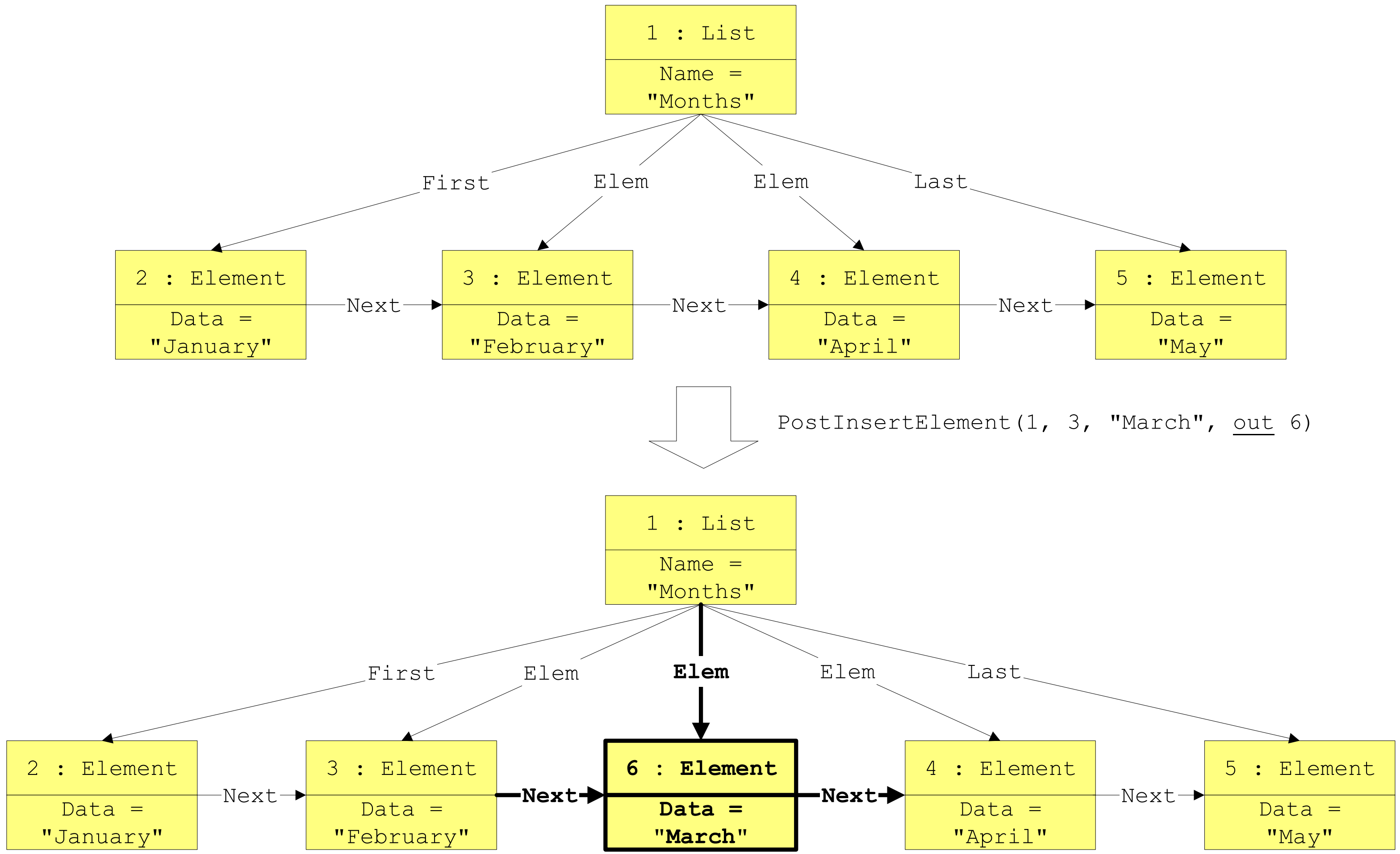
Example of a graph rewrite rule

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




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

Application of the graph rewrite rule

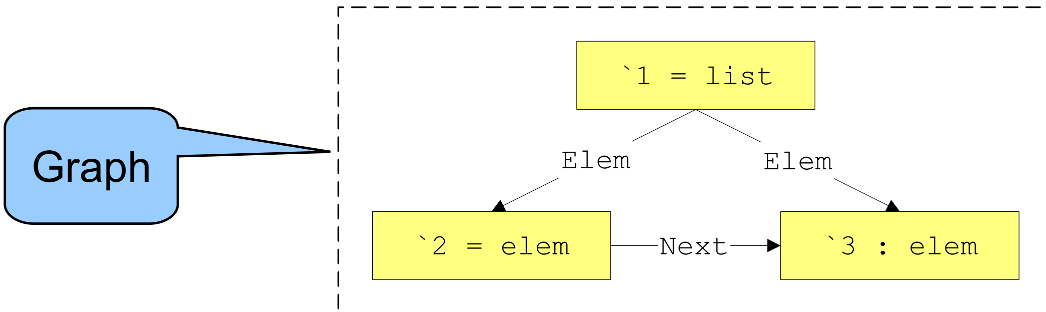


Example of a graph test

Name

Parameter

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



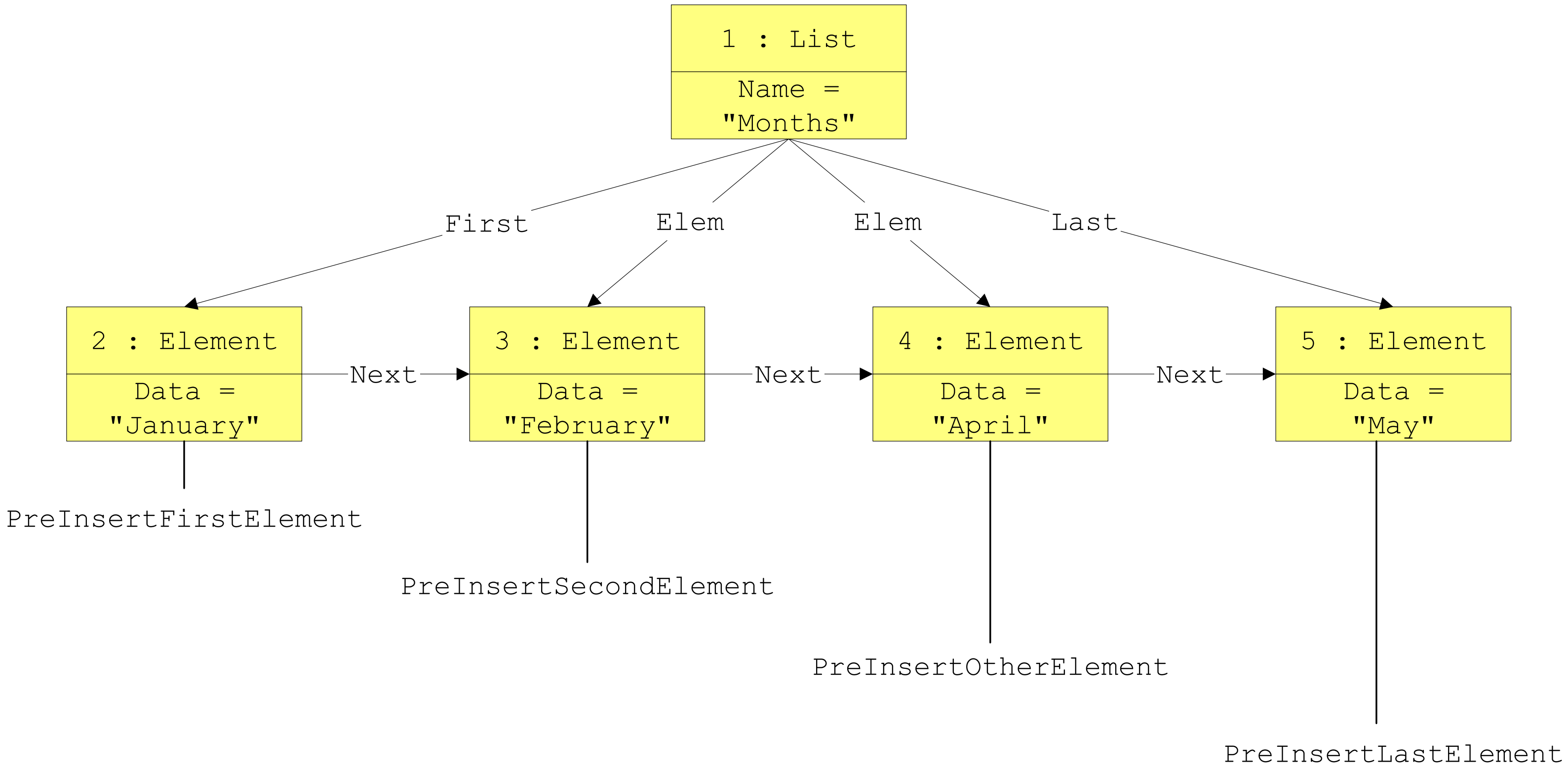

```
   return next := ´3;
end;
```

Return part

Interface of the abstract data type `List`

List creation and deletion	
`CreateList`	Creation of a list
`DeleteList`	Deletion of a list
Write operations	
`InsertFirstElement`	Insertion into an empty list
`PreInsertElement`	Insertion before an element
`PostInsertElement`	Insertion after an element
`DeleteElement`	Deletion of an element
Read operations	
`IsEmpty`	List empty?
`GetFirstElement`	First element
`GetLastElement`	Last element
`GetNextElement`	Next element
`GetPreviousElement`	Previous element
`GetData`	Data of current element

Cases of PreInsertElement



Example of programming with graph rewrite rules

Transaction

Name

Parameter

Control structure

```
transaction PreInsertElement
                  (list : List; elem : Element; data : string;
                   out new : Element) =
   choose
      PreInsertFirstElement(list, elem, data, out new)
   or
      PreInsertSecondElement(list, elem, data, out new)
   or
      PreInsertOtherElement(list, elem, data, out new)
   or
      PreInsertLastElement(list, elem, data, out new)
   end
end;
```

Theoretical Foundations

Graphs

Definition: Directed, labeled graph

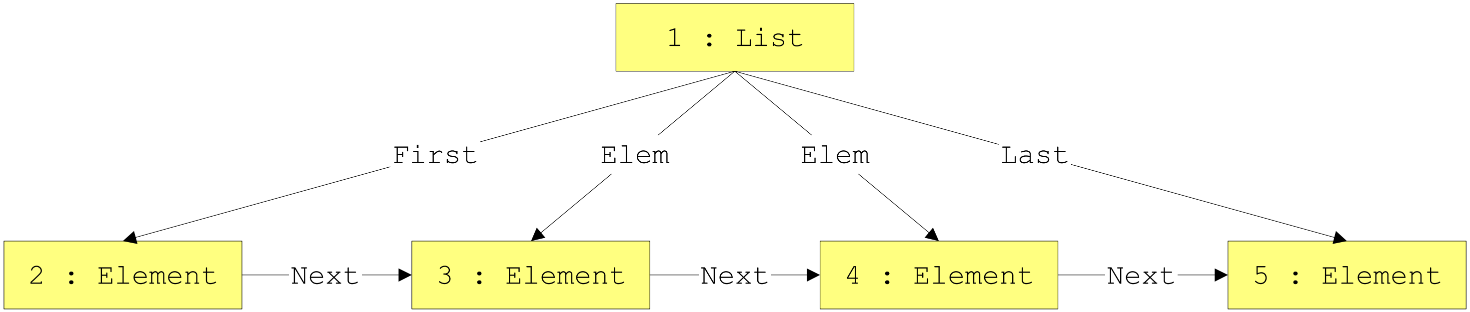
$G = (V, E, l)$ is a directed graph over label sets L_V (labels for vertices) and L_E (labels for edges) $\Leftrightarrow$

- » V is a set of nodes (node identifiers).
- » $E \subseteq V \times L_E \times V$ is a set of labeled edges.
- » $l : V \rightarrow L_V$ is a labeling function for nodes.

Remarks:

- Nodes have identifiers, but not edges.
- Thus, there are no parallel edges with the same labels.
- Edges are binary relationships.
- Nodes and edges are typed (labeled).
- So far, neither nodes nor edges are attributed.

Example of a directed graph



G = (*V*, *E*, *I*) with:

- *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)}
- *I* = { (1, List), (2, Element), (3, Element), (4, Element), (5, Element)}

Partial graphs and subgraphs

Definition: Partial graph

$G = (V, E, l)$ is a partial graph of $G' = (V', E', l') \Leftrightarrow$

- » $V \subseteq V'$, i.e., the nodes of G are also contained in G'.
- » $E \subseteq E'$, i.e., the edges of G are contained in G'.
- » $l' \mid_V = l$, i.e., the nodes of G have the same labels in G'.

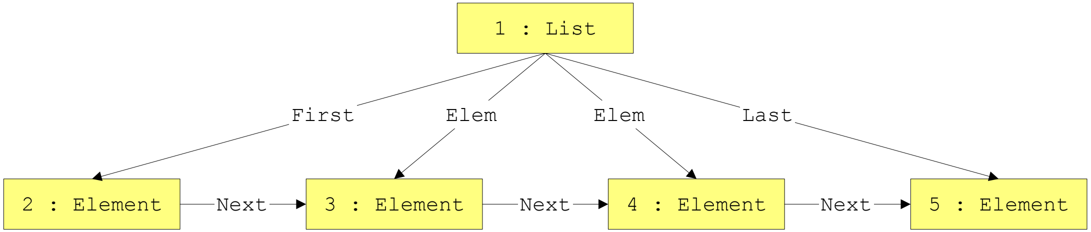
Definition: Subgraph

$G = (V, E, l)$ is a subgraph of $G' = (V', E', l') \Leftrightarrow$

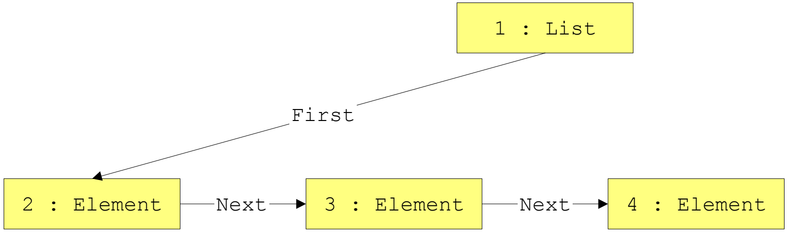
- » G is a partial graph of G'.
- » $E = \{ (v_1, el, v_2) \in E' \mid v_1, v_2 \in V \}$
  G contains all edges of G' whose sources and targets are common to G and G'.

Example of partial graphs and subgraphs

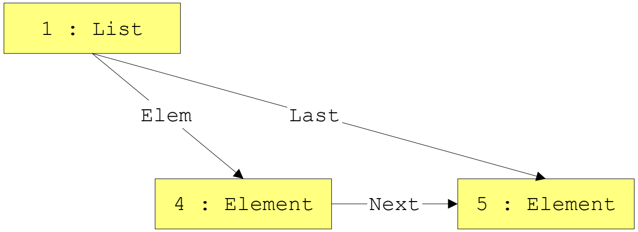
Graph



Partial graph



Subgraph



Graph morphisms

Definition: Graph morphism

A function $h : V \rightarrow V'$ is a graph morphism from G to G', i.e.,
$h : G \rightarrow G' \Leftrightarrow$

- $\forall\, v \in V : l'(h(v)) = l(v)$, i.e., labels are preserved.
- $\forall\, (v_1, el, v_2) \in E : (h(v_1), el, h(v_2)) \in E'$, i.e., edges are preserved.

Definition: Graph isomorphism

A graph morphism $h : G \rightarrow G'$ is a graph isomorphism $\Leftrightarrow$

- $h : V \rightarrow V'$ is injective and surjective
- $h : V \rightarrow V'$ induces a function $h' : E \rightarrow E'$, which must be injective and surjective, as well.

Set-theoretic graph operations (1)

Definition: Union of graphs

Let G and G' be directed graphs, $l\,|_{V \cap V'} = l'\,|_{V \cap V'}$:

- » $G \cup G' = G'' = (V'', E'', l'')$ with
  - ⇨ $V'' = V \cup V'$
  - ⇨ $\forall\, v \in V'' : l''(v) =$ <u>if</u> $v \in V$ <u>then</u> $l(v)$ <u>else</u> $l'(v)$ <u>end</u>
  - ⇨ $E'' = E \cup E'$
- » $G \oplus G'$ disjoint union of G and G':
  Rename nodes of G or G' such that $V \cap V' = \emptyset$,
  and apply the graph union defined above:
  $G \oplus G' = G \cup G'$.

Set-theoretic graph operations (2)

Definition: Difference of graphs

Let G and G' be directed graphs, $l \mid_{V \cap V'} = l' \mid_{V \cap V'}$:

- » $G \setminus G' = G'' = (V'', E'', l'')$ with
  - ⇨ $V'' = V \setminus V'$
  - ⇨ $l'' = l \mid_{V''}$
  - ⇨ $E'' = E \setminus E'$ (without deletion of dangling edges)
- » $G \setminus\setminus G' = G'' = (V'', E'', l'')$ with
  - ⇨ $V'' = V \setminus V'$
  - ⇨ $l'' = l \mid_{V''}$
  - ⇨ $E'' = (E \setminus E') \cap (V'' \times L_E \times V'') = E \cap (V'' \times L_E \times V'')$
    (with deletion of dangling edges)

Set-theoretic graph operations (3)

Definition: Intersection of graphs

Let G and G' be directed graphs, $l \mid_{V \cap V'} = l' \mid_{V \cap V'}$:

» $G \cap G' = G'' = (V'', E'', l'')$ with

⇨ $V'' = V \cap V'$

⇨ $l'' = l \mid_{V''}$

⇨ $E'' = E \cap E'$

Graph rewrite rules

Definition: Graph rewrite rule

A graph rewrite rule is a triple $r = (L, K, R)$ with:

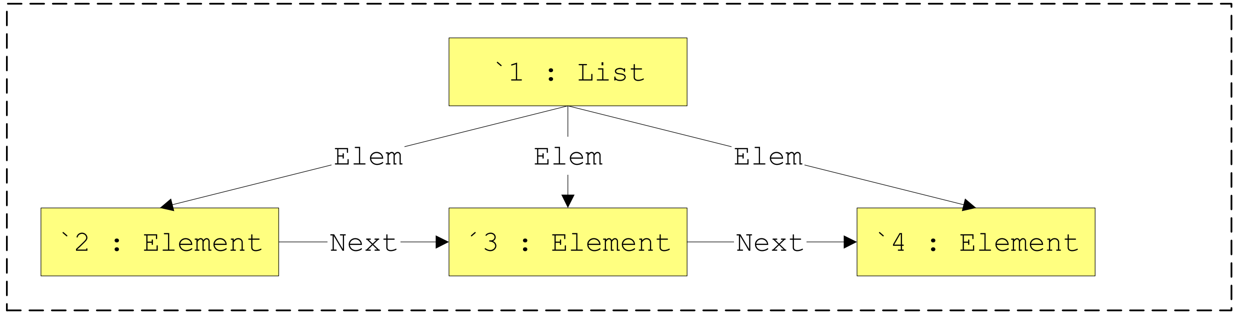
» L, the left-hand side of r, is a graph.
» R, the right-hand side of r, is a graph.
» $K = L \cap R$ is the gluing graph.

Remarks

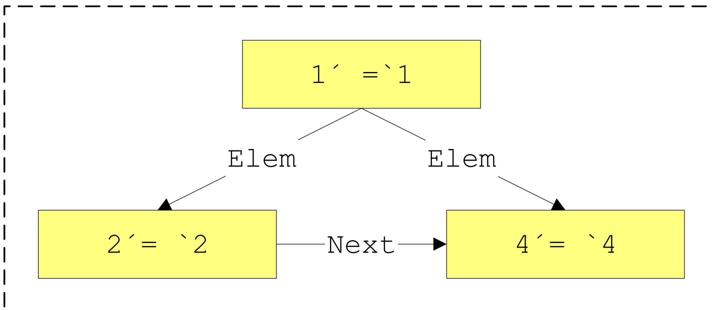
- Elements of L but not of K are deleted.
- Elements of R but not of K are inserted.
- Elements of K are preserved.
- K is called gluing graph because it is used for the embedding of new nodes of R.

Example of a graph rewrite rule (1)

<u>production</u> DeleteElement =



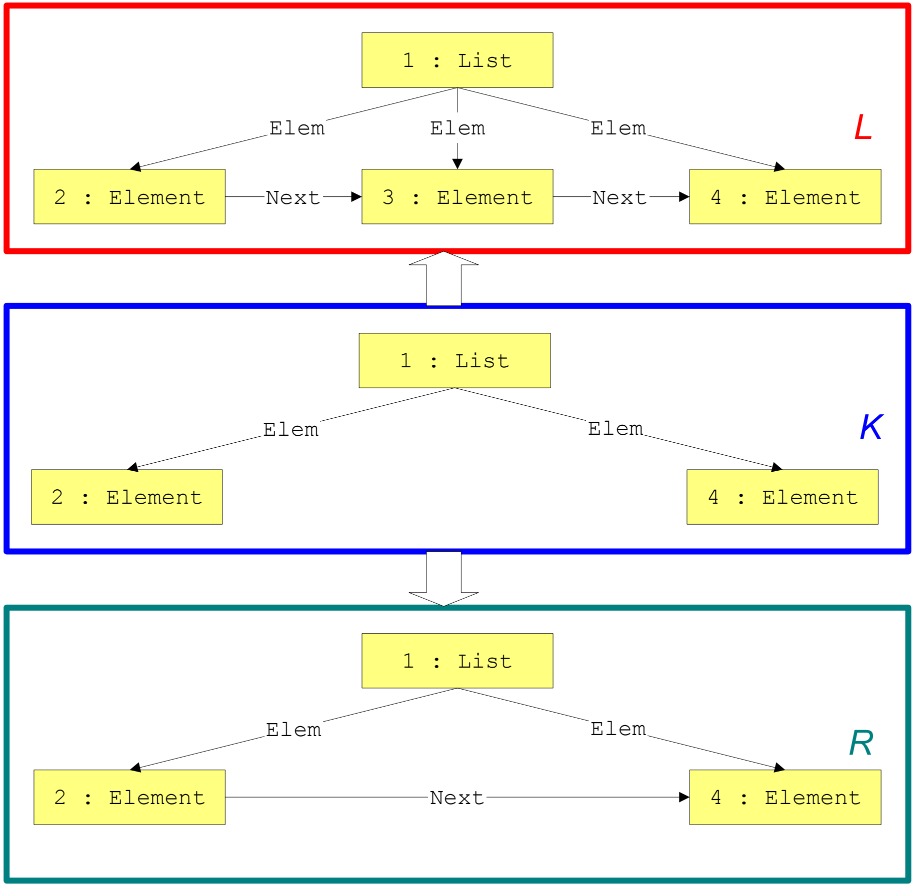
::=



<u>end</u>;

Example of a graph rewrite rule (2)

production DeleteElement =



end;

Application of a graph rewrite rule

(Direct) derivation

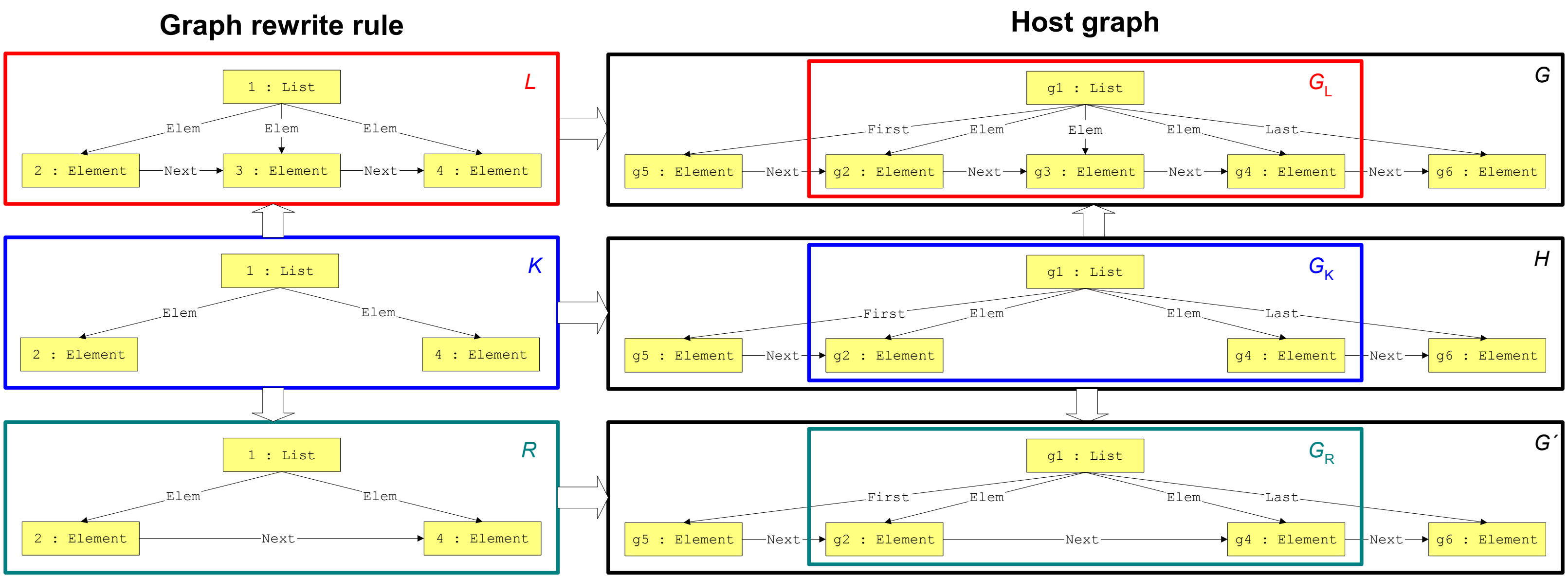
A graph $G´$ is derivable from a graph G by a rule $r = (L, K, R) \Leftrightarrow$

- There is an isomorphism $h : L \rightarrow G_L$, where G_L is a partial graph of G which determines the location of application of r.
- Nodes and edges of G_L not appearing as images of K are deleted:
  $H = G \setminus\setminus (h(L) \setminus h(K))$
- Nodes and edges of R which do not belong to K are inserted:
  $G´ = H \oplus h´(\mathrm{R} \setminus K)$, where
  - ⇨ $h´$ maps nodes of $R \setminus K$ such that they do not occur in H,
  - ⇨ Context edges with sources or targets from K are transferred to the respective nodes in $h(K)$.

Other variants of graph rewrite rules

- *h* need only be a morphism ⇒
  - Higher flexibility by identification of graph elements
  - Danger of undesired effects of rule applications
- G_L must even be a subgraph G ⇒
  - Larger left-hand sides
  - Larger rule sets
- No edges from deleted nodes to context nodes (Dangling Edge Condition) ⇒
  - Exclusion of undesired side effects
  - Large left-hand sides
- Empty gluing graph ⇒
  - Embedding of nodes of the right-hand side must be specified explicitly by embedding rules

Example of the application of a graph rewrite rule



Graph rewriting systems

Definition: Graph rewriting system

A graph rewriting system is a tuple
$gs = (L_V, L_E, R, S)$ with:

- L_V : finite set of node labels
- L_E : finite set of edge labels
- R : finite set of rules $r = (L, K, R)$ (L, K, R graphs over L_V, L_E)
- S : start graph (over L_V and L_E)

Definition: Derivability

Let $gs = (L_V, L_E, R, S)$ be a graph rewriting system. A graph G is derivable from the start graph S using the rule set R $gs \Leftrightarrow$
There is a sequence $G_1 \ldots G_n$ with
$S \rightarrow G_1 \ldots \rightarrow G_n = G$

Proof techniques

Proof by induction

A predicate *p* is proved for all derivable graphs as follows *(n* : length of derivation)*:*

1. $n = 0$: *p* holds for the start graph *S*.
2. $n \rightarrow n + 1$: Let *G* be a graph which may be derived in *n* steps from the start graph.
   Induction assumption: *p* holds for *G*.
   Induction conclusion: *p* holds for all graphs *G´* which may be derived from *G* by some rule *r* (in one derivation step).
   (Induction conclusion has to be proved for all rules *r* and all potential locations of application.)

Examples of (provable) properties of list graphs

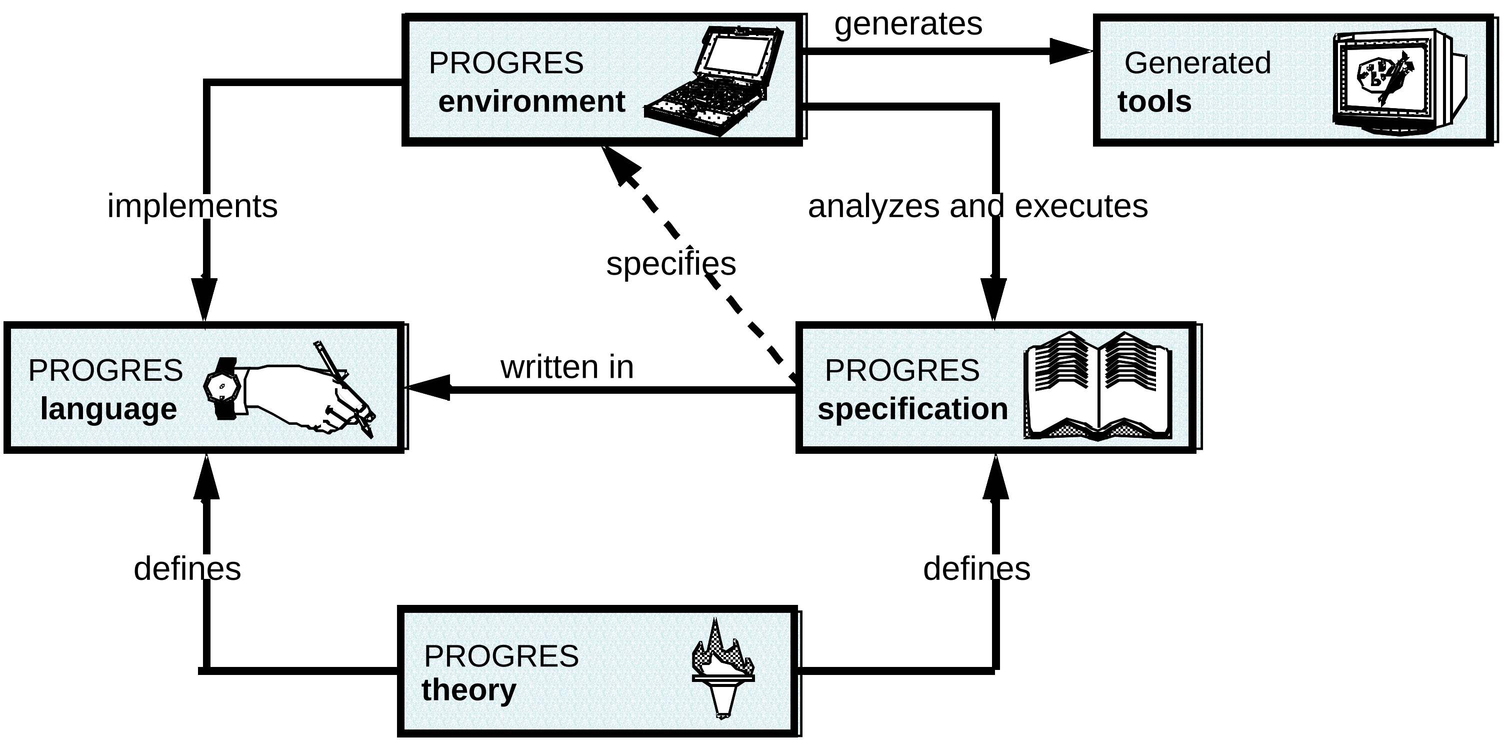
- Each list has at most one `First` element.
- Each list has at most one `Last` element.
- The `First` element does not have a predecessor.
- The `Last` element does not have a successor.
- Each element has at most one predecessor.
- Each element has at most one successor.
- ...

Specification with PROGRES

What is PROGRES?

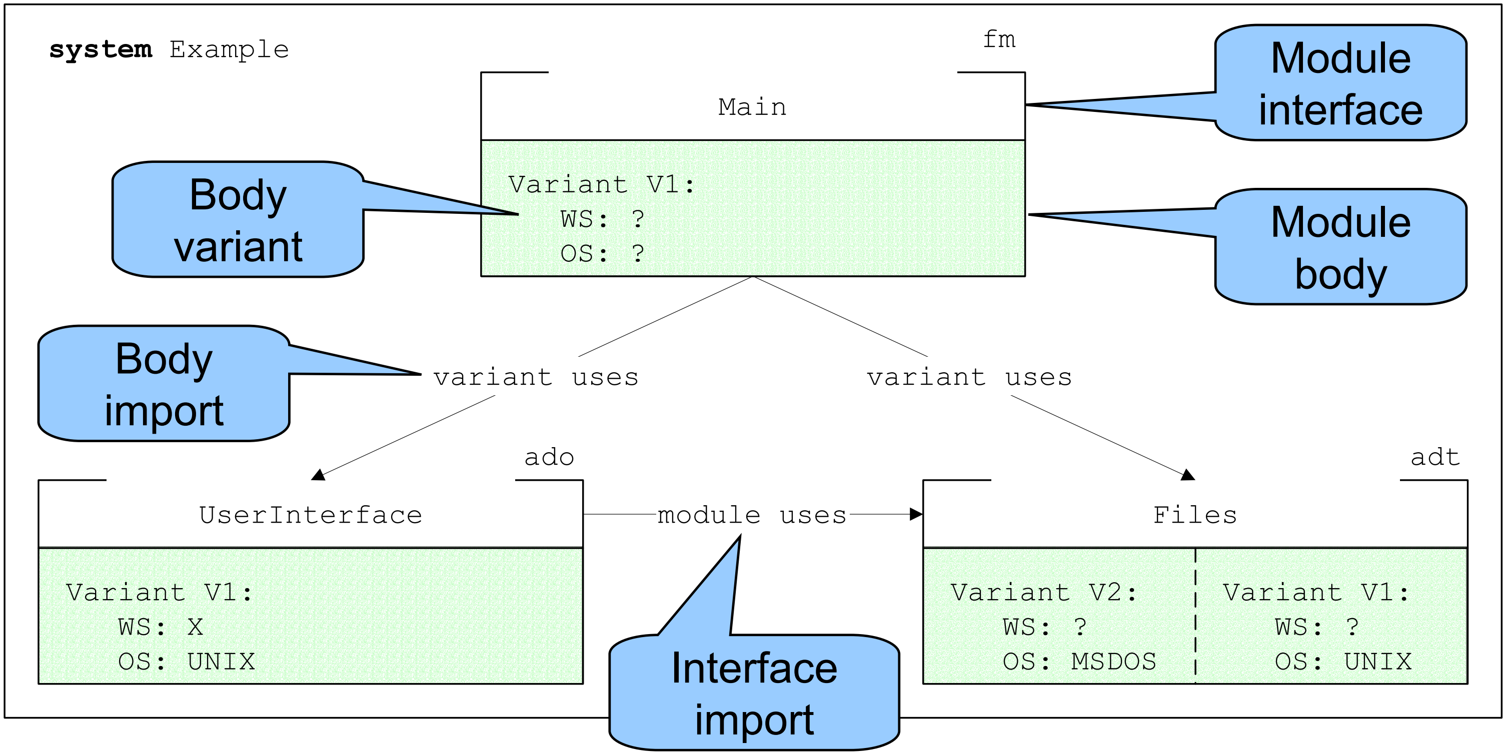
- **PROGRES** = **PRO**grammed **GR**aph R**E**writing **S**ystems
- Multiparadigmatic **Specification language**, based on graph rewriting
  - » Object-oriented modeling of graph schemata
  - » Declarative definition and incremental evaluation of derived attributes
  - » Rule-based and visual specification of graph tests and graph transformations
  - » Imperative and non-deterministic programming
- **Development environment** for the construction of specifications
  - » Syntax-aided editor
  - » Static analyses
  - » Interpreter
  - » Code generator

Components of PROGRES



Example

Tools for programming-in-the-large



Explanations

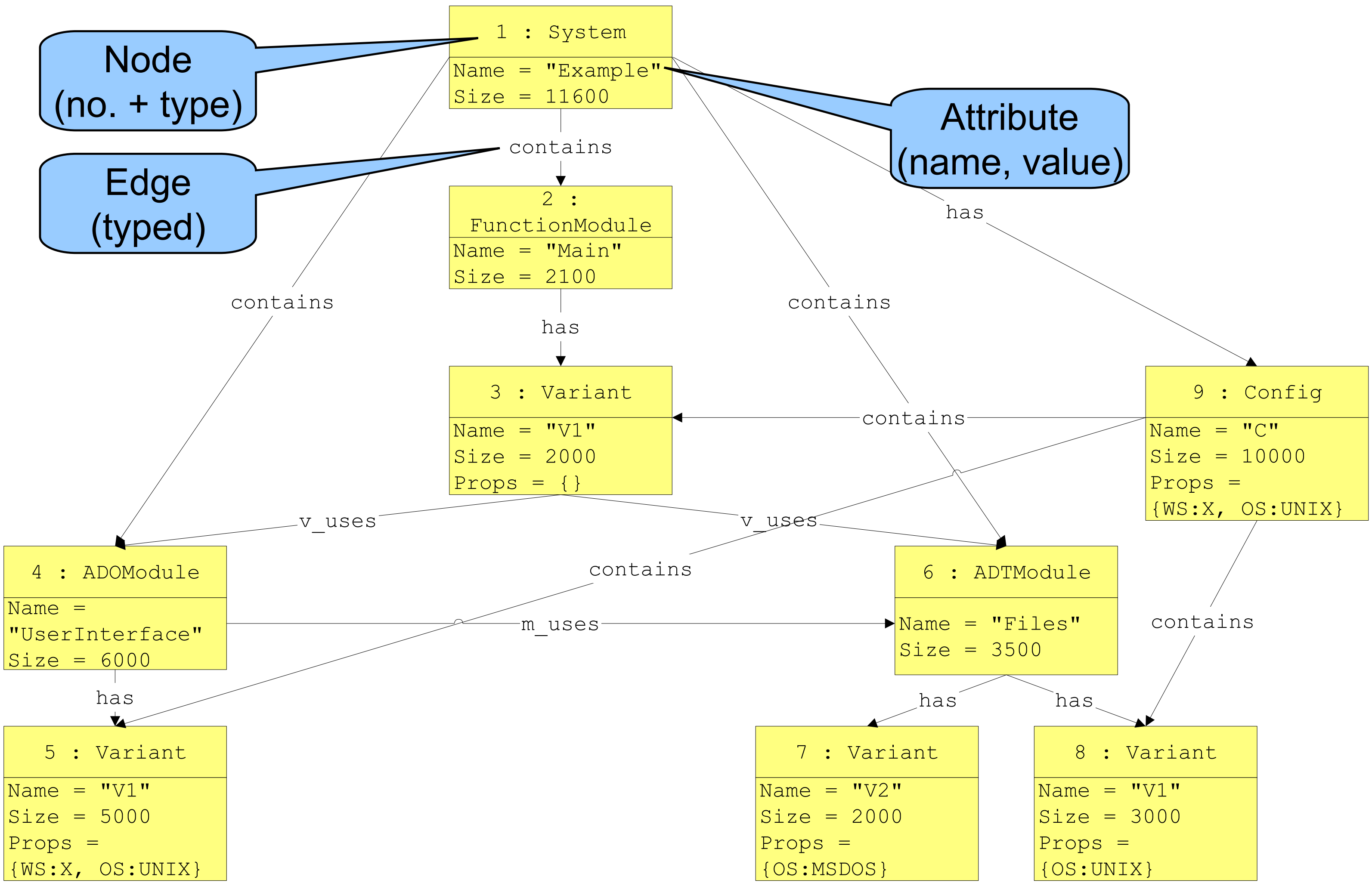
- There are three types of modules (fm, ado, adt).
- The function module (fm) `Main` exports a set of functions; it has exactly one body variant.
- This variant imports the modules `UserInterface` and `Files`.
- The data object module (ado) `UserInterface` realizes a data store and its access operations (in exactly one variant).
- The interface of module `UserInterface` imports from the module `Files` the type `File`.
- The data type module (adt) `Files` exports a data type with operations for creating and manipulating an arbitrary number of instances; it has two variants.
- Properties of variants are defined through a set of attributes (`WS` for Window System and `OS` for Operating System).
- Attributes either have concrete values (like `UNIX` or `MSDOS`) or are undefined.

Desired tool functionality

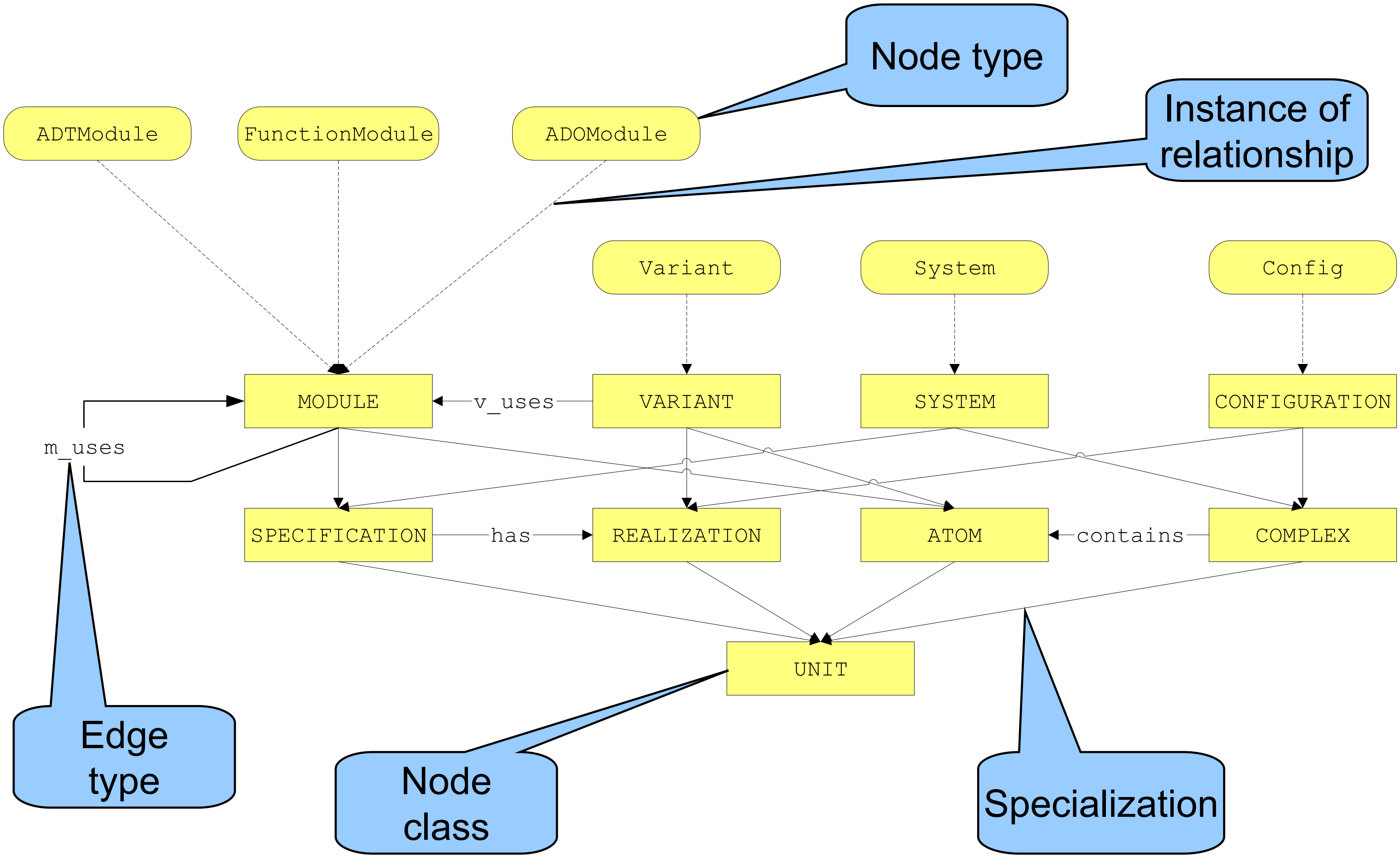
- Syntax-aided editing of module graphs (context free correctness is enforced, e.g. variant import do not emanate from module interfaces)
- Checking of context sensitive constraints (e.g., no import cycles)
- Completely automatic or semi-automatic configuration of system variants satisfying given properties
(Attention: Requires consistent selection of variants. In the example, there is only one consistent configuration for `WS = X` and `OS = UNIX`.)

Attributed Graphs and Graph Schemata

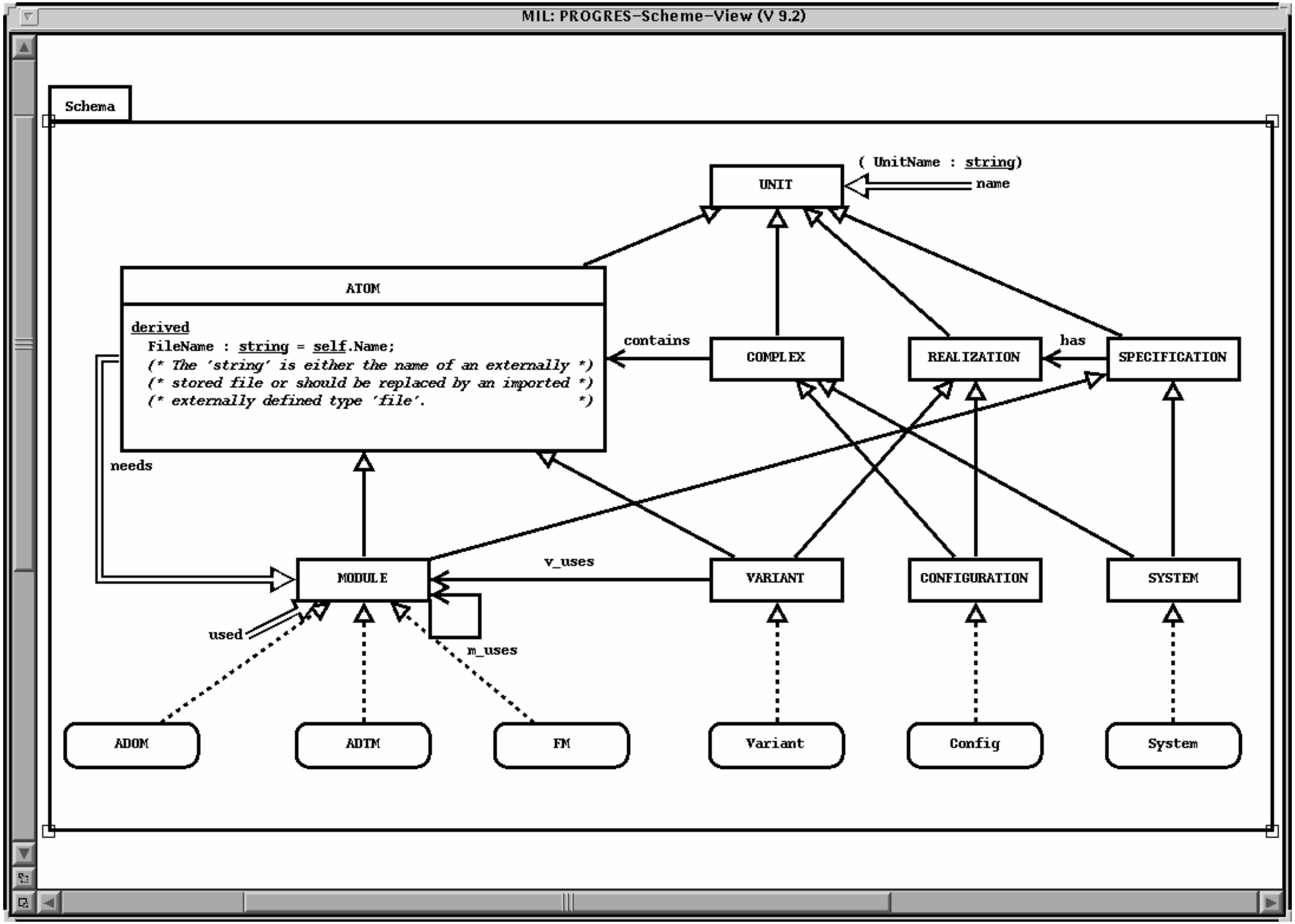
Representation of the example as an attributed graph



Graphical definition of a graph schema



PROGRES schema editor



Definition of attributes

- A graph schema may be specified both textually and graphically.
- Attributes and attribute evaluation rules may be defined in a textual (sub-)view.
- Intrinsic attributes receive their values by explicit assignment and may be initialized with a constant.
- Derived attributes are calculated from the attribute values of related nodes with the help of a (directed) equation.
- Related nodes are all nodes which may be reached via a path, e.g.:
  - `self` : Returns the current node
  - `-e->` and `<-e-`, respectively : Traversal of edges of type `e` (in forward and backward direction, respectively)
  - Concatenation : `p1 & p2`

Textual definition of a graph schema (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    *)
```

Node class

Derived attribute

Intrinsic attribute

Redefined evaluation rule

Textual definition of a graph schema (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'. *)
```

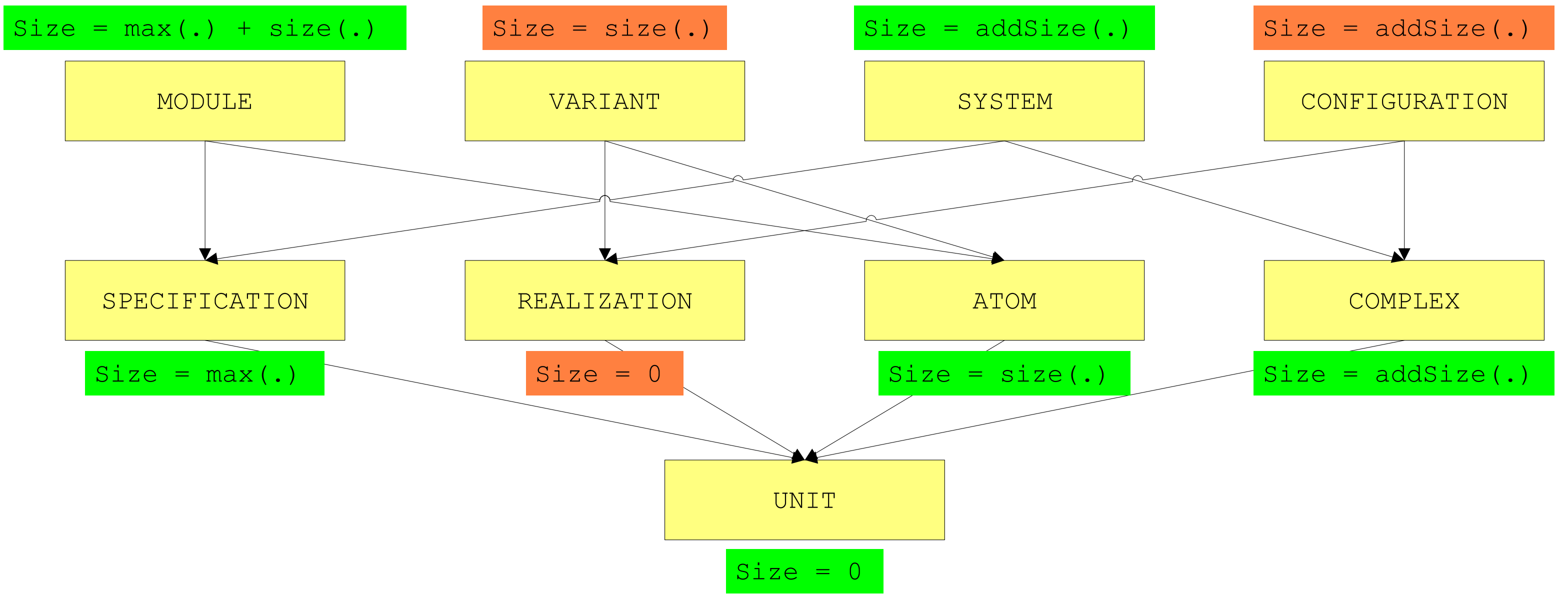
Edge type

Cardinality

Textual definition of a graph schema (3)

```
node class SYSTEM is a SPECIFICATION, COMPLEX
   redef derived
      Size = addSize( 0, all self.-contains-> );
           (* Resolves inheritance conflict of attribute   *)
           (* definitions in 'SPECIFICATION' and 'COMPLEX' *)
           (* by preferring definition in 'SPECIFICATION'. *)
end;

node class CONFIGURATION is a REALIZATION, COMPLEX end;
    (* A 'CONFIGURATION' is a set of variants of module    *)
    (* realizations which fulfill all required properties. *)

node class MODULE is a SPECIFICATION, ATOM
  redef derived
     Size = size( self.File ) + max( 0, all self.-has->.Size );
          (* Resolves inheritance conflict of attribute    *)
          (* definitions in 'SPECIFICATION' and 'COMPLEX'  *)
          (* by building the sum of both definitions which *)
          (* are in conflict to each other.                *)
end;      (* A 'SYSTEM' contains a set of 'MODULES' which  *)
          (* have 'VARIANTS' as their realizations.        *)

node class VARIANT is a REALIZATION, ATOM end;
```

Attribute redefinition conflicts



Explicitly defined evaluation rule

Inherited evaluation rule

Path expressions and restrictions

- A **path expression** is
  - » a derived relation between nodes (edges are intrinsic relations):
    v_1 $=p\Rightarrow$ $v_2 \Leftrightarrow$ There is a path p from v_1 to v_2
  - » a function on node sets:
    $p(V) = \{ v_2 \mid \exists\, v_1 \in V : v_1 =p\Rightarrow v_2 \}$
- A **restriction** is a “unary path expression", i.e., a set of nodes is restricted to those elements which satisfy a certain condition.
- Path expressions and restrictions may be specified:
  - » textually
  - » graphically

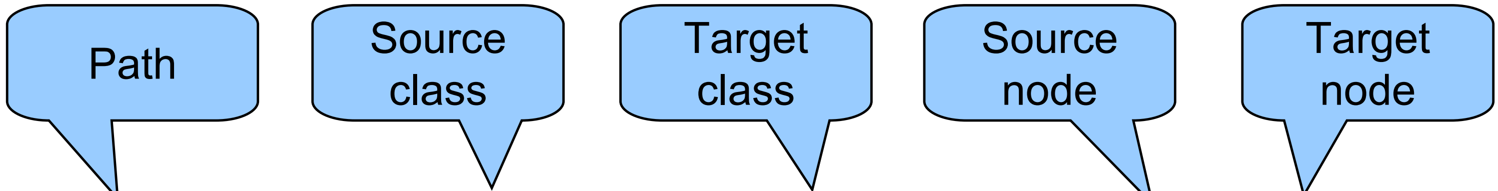
Textual path expressions

Operator	**Semantics**
p & *q*	Concatenation of *p* and *q*
p <u>or</u> *q*	Connection by *p* or *q*
p <u>and</u> *q*	Connection by *p* and *q*
p <u>but not</u> *q*	Connection by *p* but not by *q*
[*p* | *q*]	Connection by *p*, else by *q*
p+	Transitive closure
*p**	Reflexive and transitive closure
{*p*}	Maximal iteration

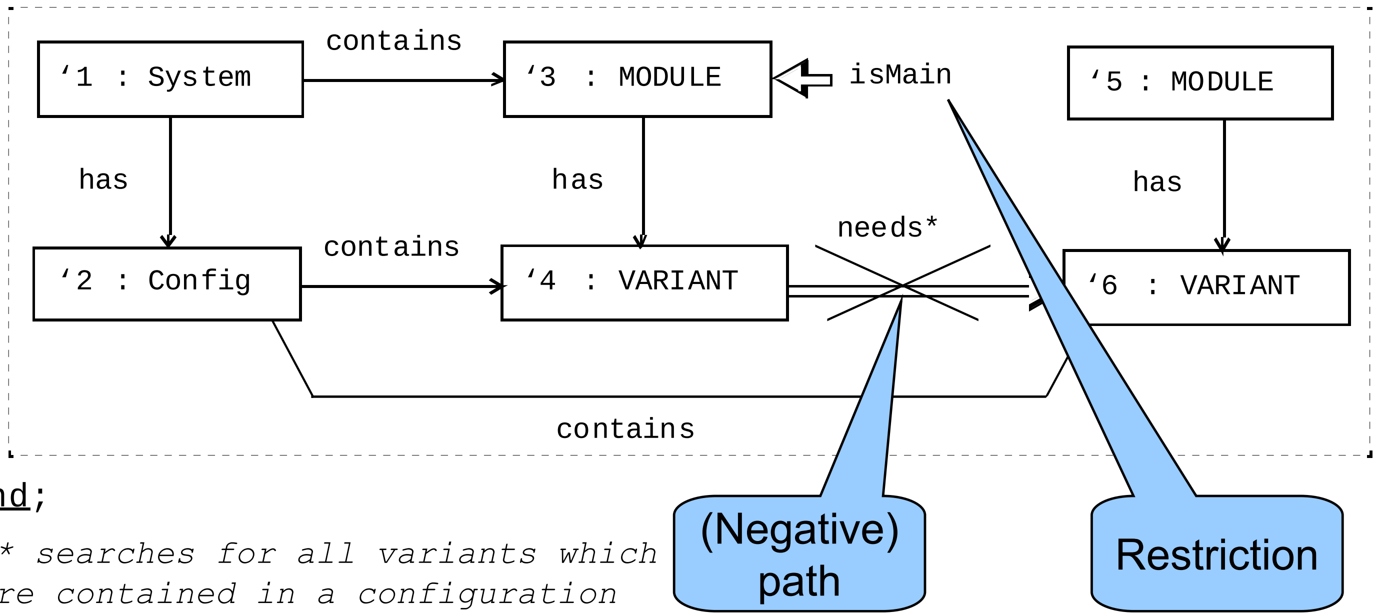
Examples of textual path expressions

```
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;
```

Example of a graphical path expression




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




```
end;
```

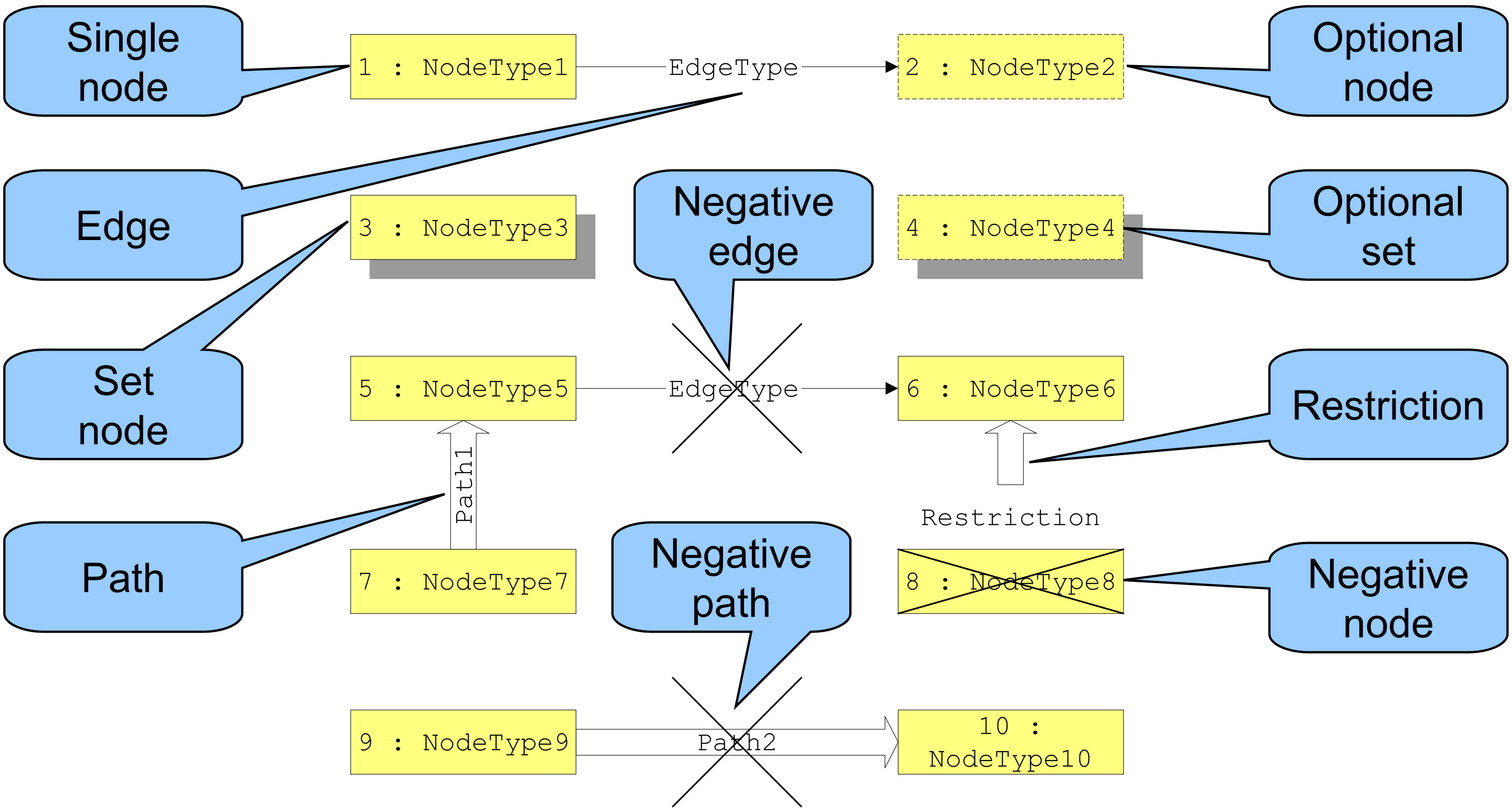
(searches for all variants which are contained in a configuration but are not reachable from the top-level module *)*

Graph Rewrite Rules

Composition of graph rewrite rules

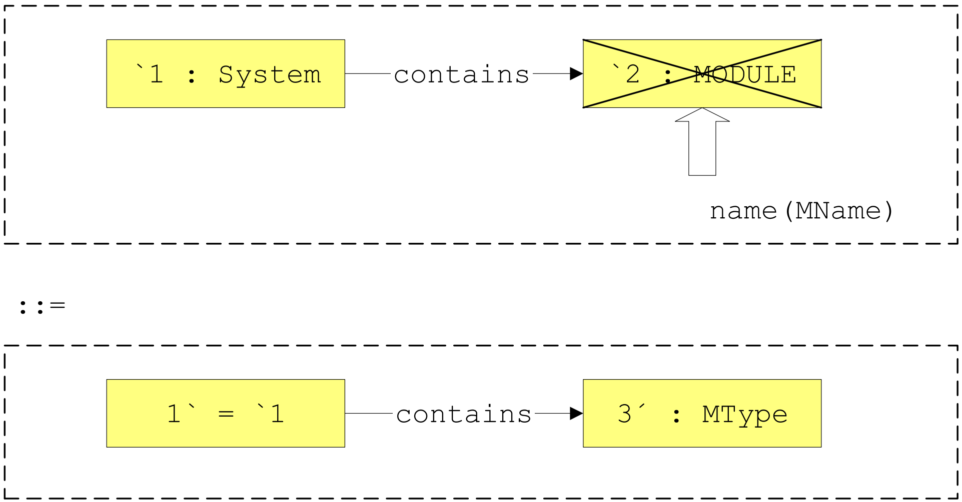
```
production P ( parameter list ) =

   Left-hand side with:
      - single nodes, set nodes, negative/optional nodes
      - positive and negative edges between pairs of nodes
      - positive and negative paths between pairs of nodes
      - node restrictions

   ::=

   Right-hand side with:
      - single nodes, set nodes, optional nodes
      - edges between pairs of nodes

   fold      Specified nodes may be identified;
   condition Conditions on attributes of left-hand side nodes;
   embedding Embedding rules for nodes of right-hand side;
   transfer  Attribute assignments for nodes of right-hand side;
   return    Assignments to out parameters;
end;
```

Elements of left-hand sides and right-hand sides



Example for parameters, negative nodes, attribute transfer, etc.

<u>production</u> CreateModule(MName : <u>string</u>; InterfaceDescription : file;
MType : <u>type in</u> MODULE; <u>out</u> NewM : MODULE) =

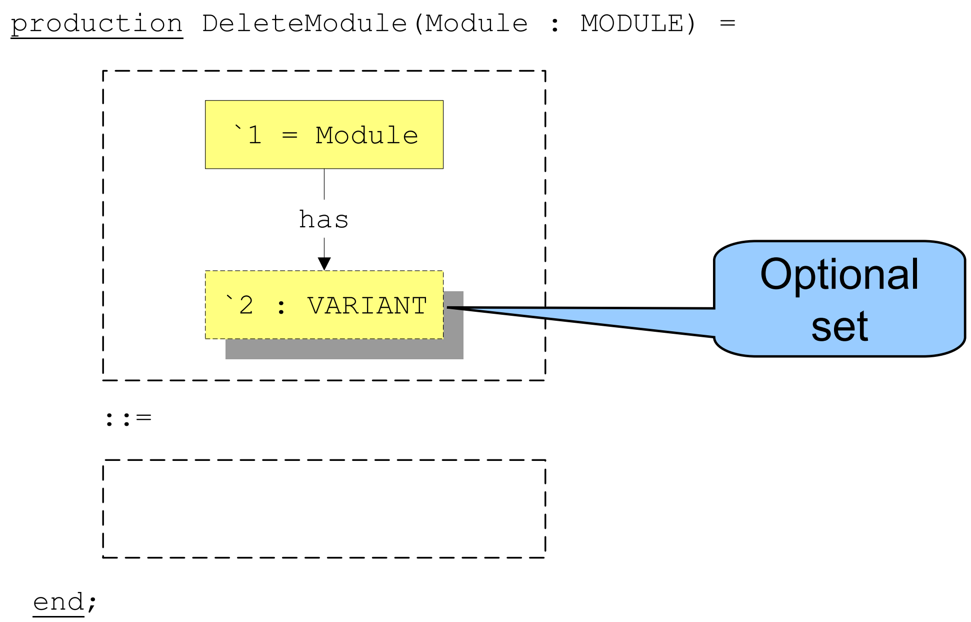


(Creates a module with name ´MName´if the ´System´ does not already contain a module with this name. The ´Mtype´ parameter is a ´FunctionModule´ or ´ADTModule´ or ´ADOModule´. *)*

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

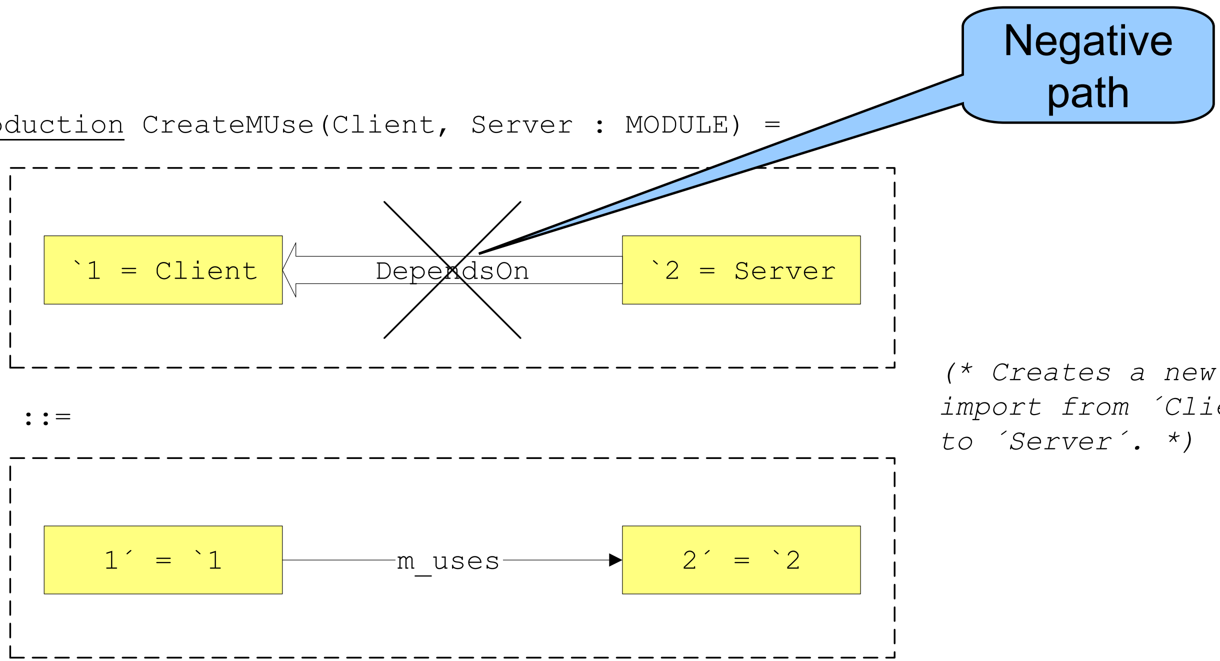
<u>restriction</u> name(UName : <u>string</u>) : UNIT = <u>valid</u> (<u>self</u>.Name = UName) <u>end</u>;
(The restriction is valid if the current ´UNIT´ is named ´UName´. *)*

Example of optional set nodes



Example of a negative path

```
production CreateMUse(Client, Server : MODULE) =
```



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

```
end;
```

Embedding rules

Rule	**Semantics**
`copy -e-> from ` `` `n `` ` to m´`	Copy outgoing `e` edges from node `` `n `` of the left-hand side to node `m´` of the right-hand side
`remove -e-> from ` `` `n ``	Delete outgoing `e` edges from node `` `n ``
`redirect -e-> from ` `` `n `` ` to m´`	Redirect outgoing `e` edges from `` `n `` to `m´`
`<-e-` instead of `-e->`	Analogously for incoming rather than outgoing edges
`-e-> as -f->` instead of `-e->`	Relabeling
`-e-> as <-f-` statt `-e->`	... and change of direction

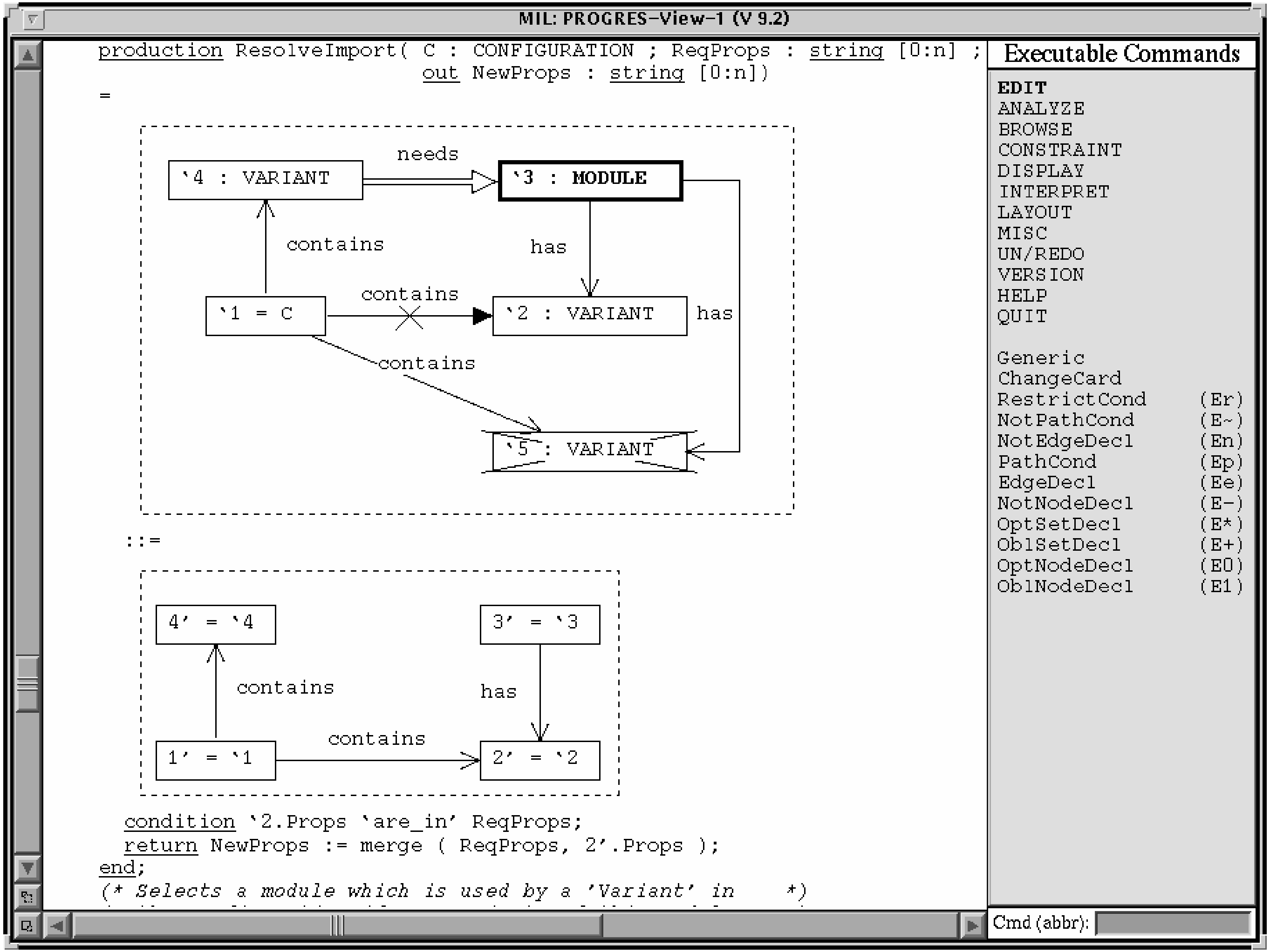
Example of embedding rules

```
production ChangeModuleType(Module : MODULE; NewType : type in MODULE;
                            out NewNode : NewType) =

     `1 = Module

   ::=

     1´: NewType

   embedding redirect -m_uses->, -has->, <-contains-,
                      <-m_uses-, <-v_uses- from `1 to 1´;
   transfer 1´.Name := `1.Name;
            1´.File := `1.File;
   return NewNode := 1´;
end;
```

(Replaces given module by a module of type ´NewType´with same edges and attribute values. Unfortunately, this has to be handled by deletion and re-insertion. *)*

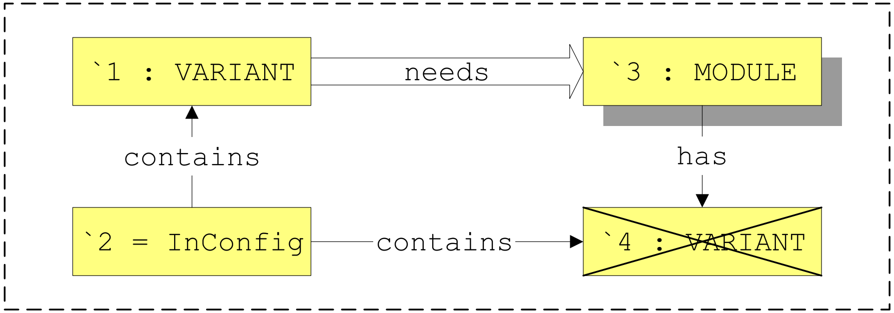
Embedding rule

PROGRES rule editor



Example of a graph test

```
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. *)
```

Searching for subgraphs

- Complexity of naive implementation: $o(n^k)$
  (for each of *k* nodes in *L* there are *n* candidates in *G*).
- Heuristics (sketch):
  - » Start at those nodes which are fixed by input parameters.
  - » Extend the match by nodes which may be determined in a unique way (incoming or outgoing edges of cardinality 1).
  - » Process remaining candidate sets by increasing cardinality.
  - » Process set nodes at the end.

Control Structures

Control structures: motivation and properties

- Composition of graph tests into complex **queries** (which do not modify the host graph)
- Composition of graph rewrite rules (and graph tests) into complex **transactions**
- **ACID** properties of transactions (and graph rewrite rules):
  - **A**tomic: either complete execution or no modification of the host graph
  - **C**onsistent: consistency-preserving transformation
  - **I**solated: isolation in multi-user mode
  - **D**urable: persistent
- Additional property: **non-determinism**
- Failure of executing an operation results in **backtracking**

Overview of control structures

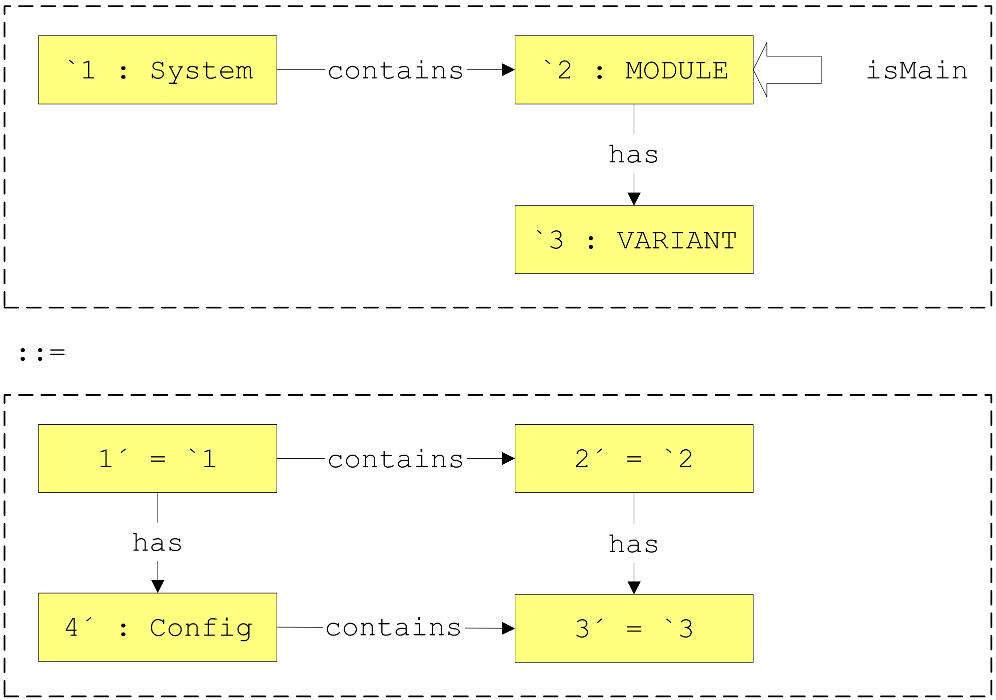
Control structure	Semantics
`p & q`	Sequence
`p and q`	`p` und `q` in any order
`p or q`	Non-deterministic choice
`choose p1 else p2 ... end`	Try `p1`, else `p2` ...
`loop p end`	Loop (execute `p` as long as possible)
`for all n do p end`	Execute `p` for all nodes `n`
`use v : ... do p end`	Block with declaration of variables

Example of a transaction with backtracking

```
transaction CreateConfig(CName : string; CProps : string [0:n]) =
   use ReqProps := CProps do
      InitConfig(CName, ReqProps, out ReqProps)
    & loop
         ResolveImport(CName, ReqProps, out ReqProps) (* see p. 56 *)
      end
    & not UnresolvedImportExists(CName) (* similarly to p. 57 *)
   end
end;
```

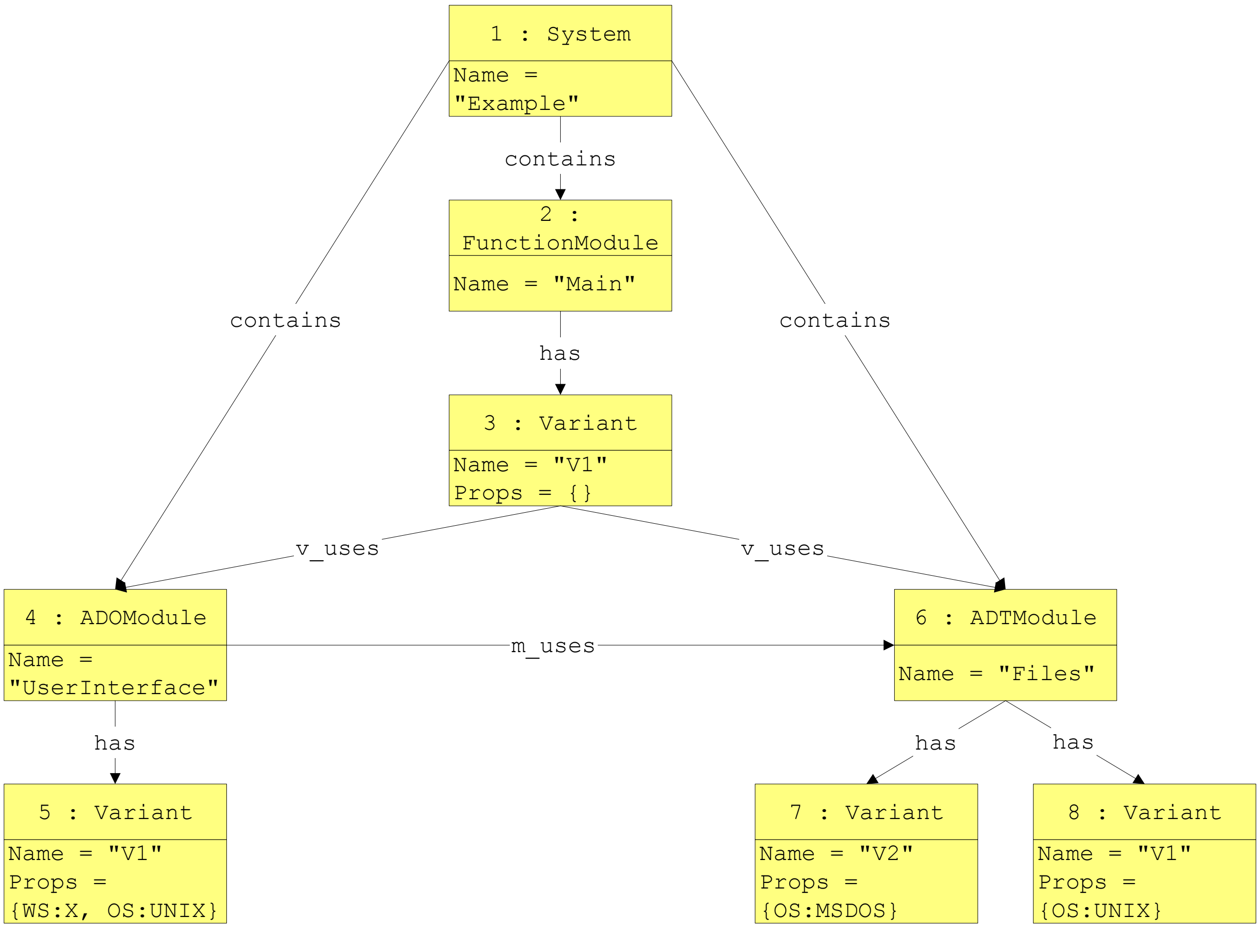
Rule for initializing a configuration

```
production InitConfig(CName : string; CPropsIn : string; out CPropsOut : string) =
```



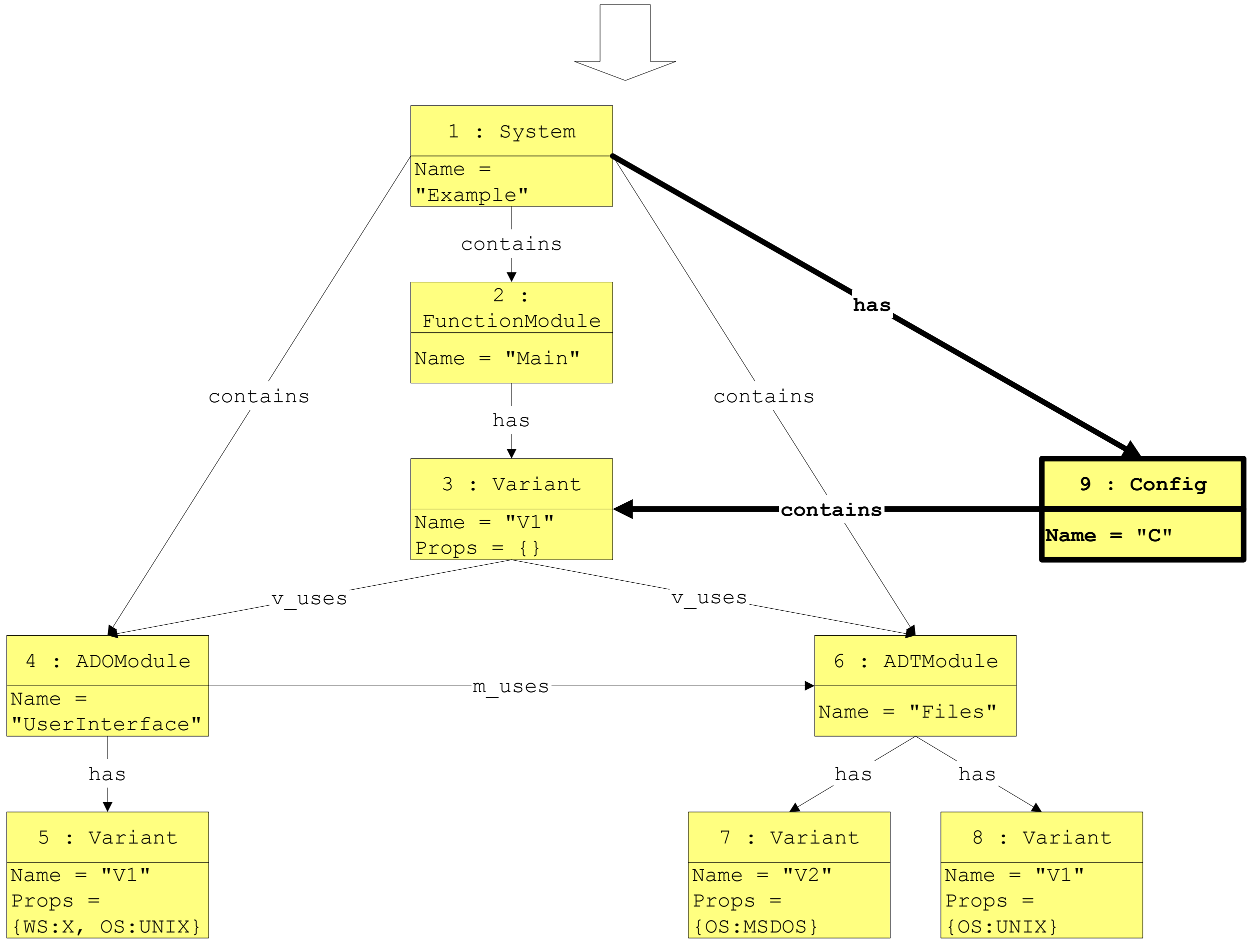

```
   condition `3.Props in CPropsIn;
   transfer 4´.Name := CName;
   return CPropsOut := merge(`3.Props, CPropsIn);
end;
```

Example (1)


1 : System
Name = "Example"
contains
2 : FunctionModule
Name = "Main"
has
3 : Variant
Name = "V1"
Props = {}
contains
contains
v_uses
v_uses
4 : ADOModule
Name = "UserInterface"
m_uses
6 : ADTModule
Name = "Files"
has
has
has
5 : Variant
Name = "V1"
Props = {WS:X, OS:UNIX}
7 : Variant
Name = "V2"
Props = {OS:MSDOS}
8 : Variant
Name = "V1"
Props = {OS:UNIX}

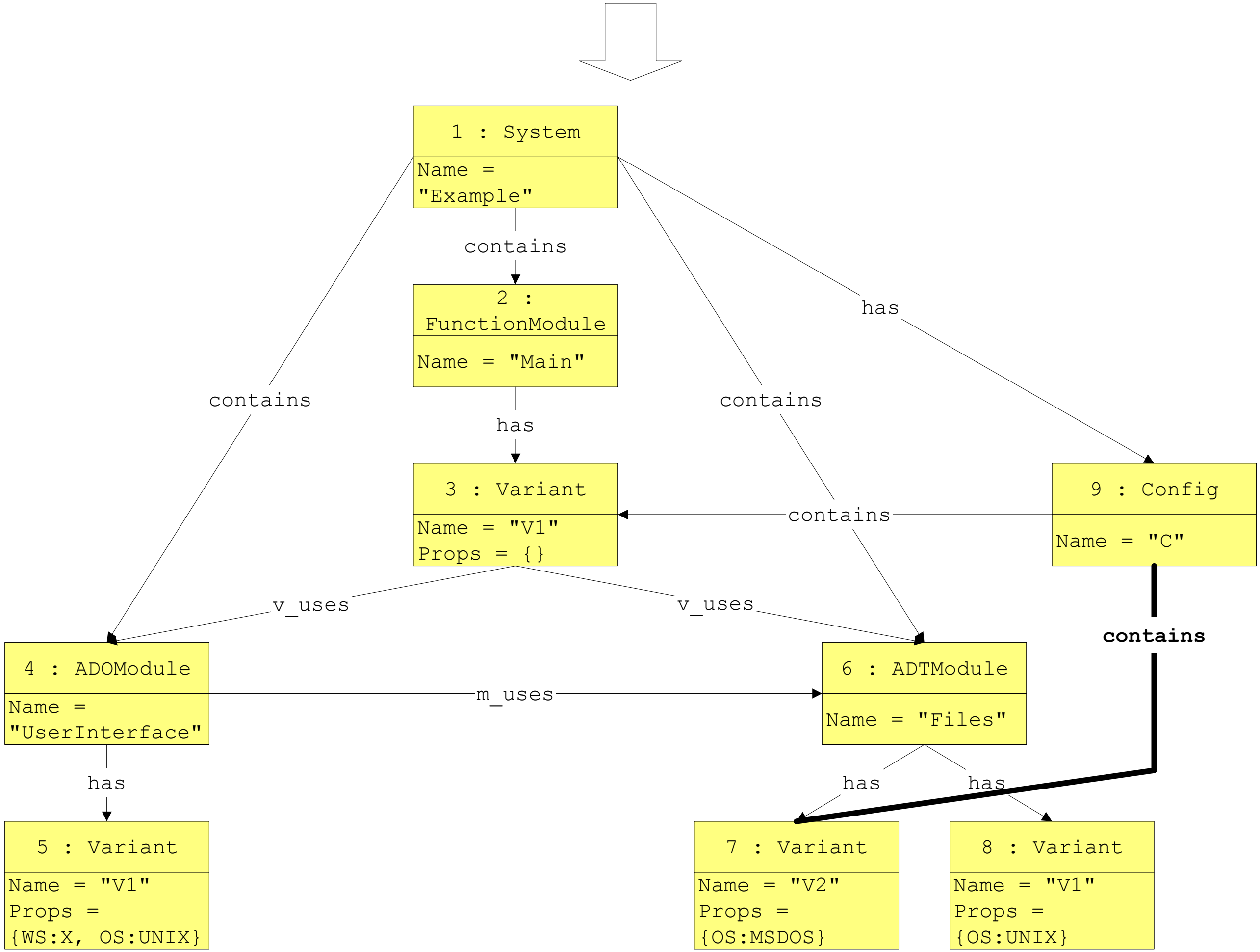
Example (2)

InitConfig("C", {}, <u>out</u> ReqProps = {})



Example (3)

ResolveImport("C", {}, out ReqProps = {OS:MSDOS})

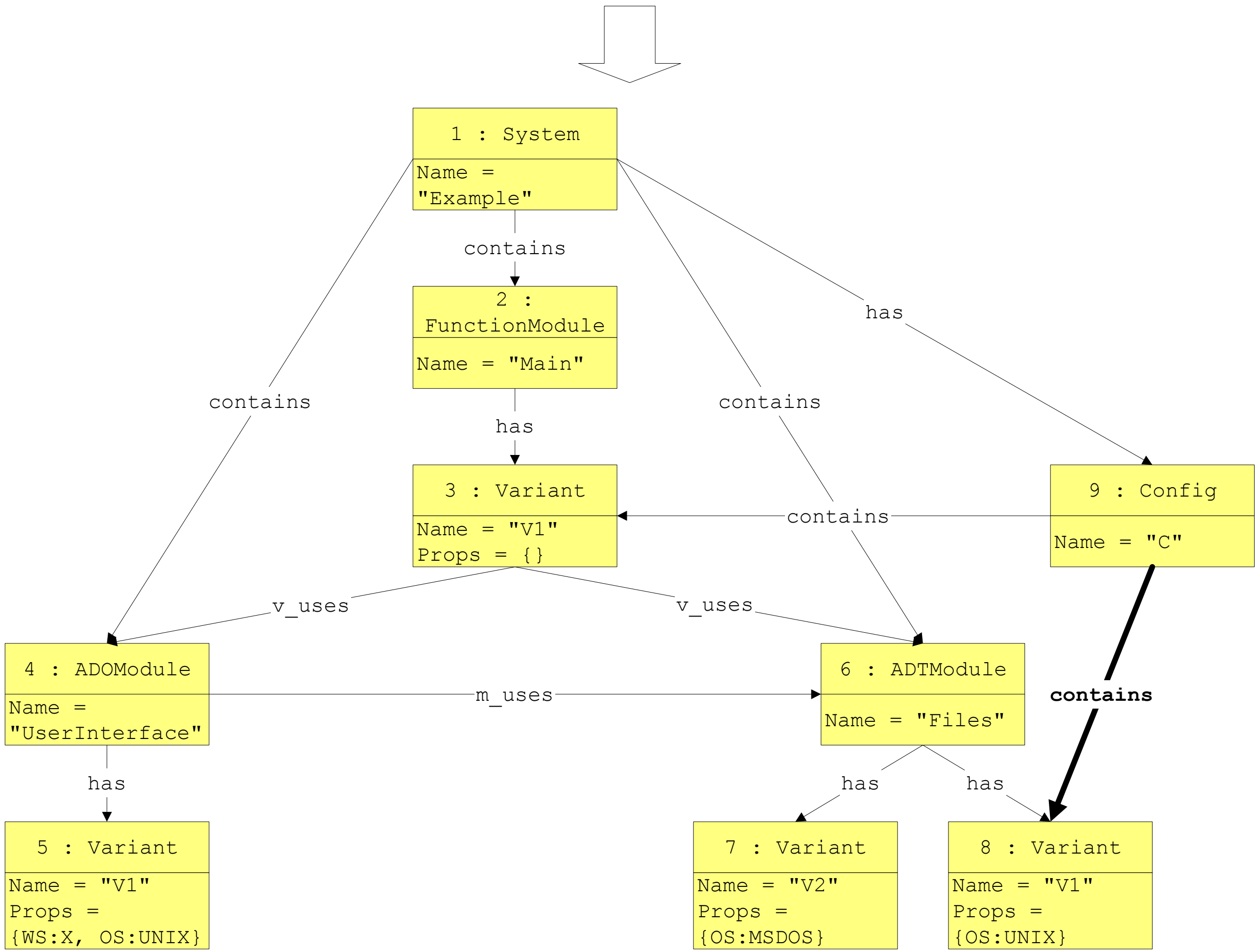


Example (4)

- `ResolveImport` fails for the module `UserInterface` because `{OS:UNIX}` <u>`in`</u> `{OS:MSDOS}` does not hold.
- Loop terminates successfully, but the subsequent test `UnresolvedImportExists` reveals an unresolved import.
- As a result of backtracking, the previous selection of the variant of `Files` is revised (slide 68 shows the situation after selection of another variant).
- Finally, a variant of `UserInterface` may be selected successfully (slide 69).

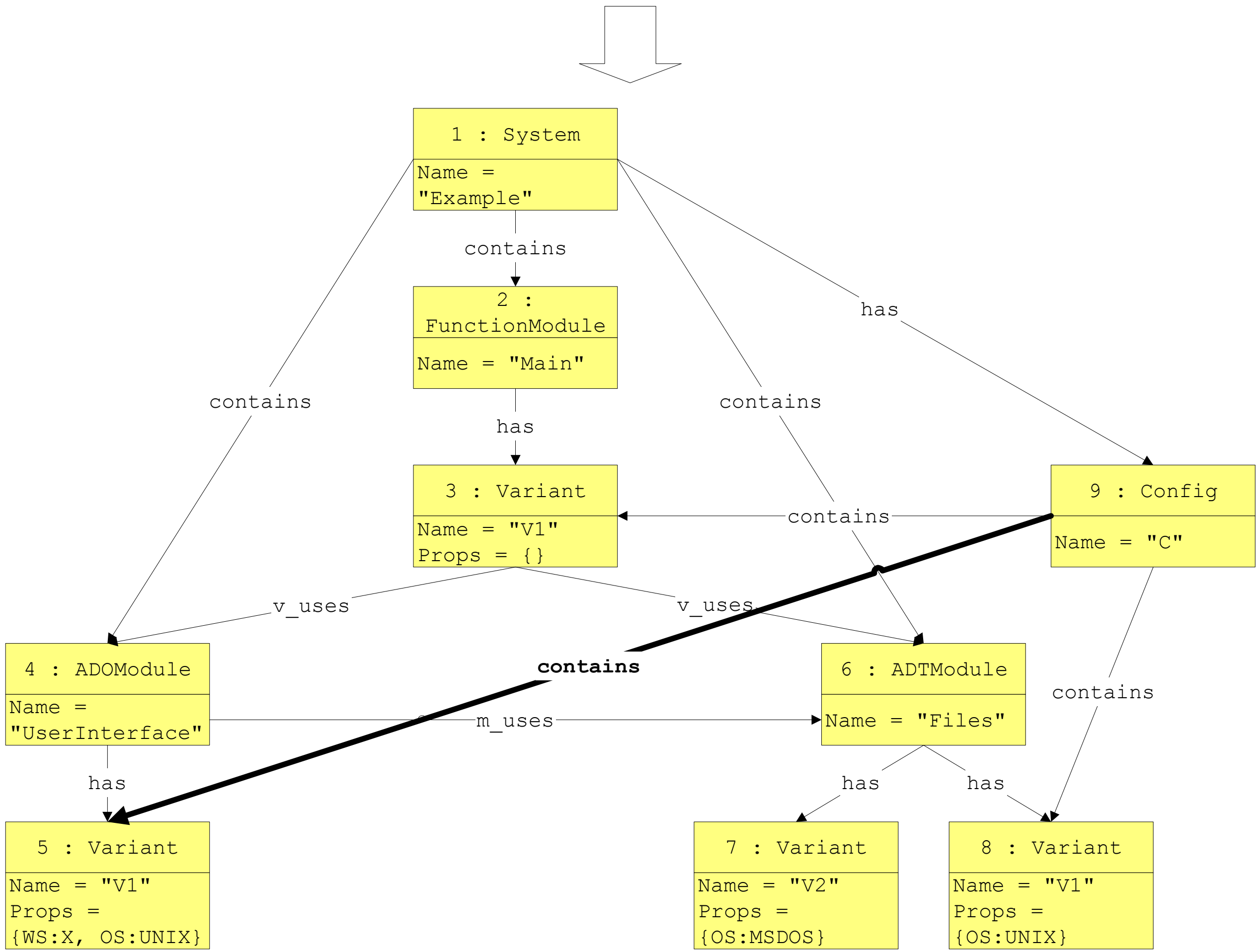
Example (5)

ResolveImport("C", {}, <u>out</u> ReqProps = {OS:UNIX})



Example (6)

ResolveImport("C", {OS:UNIX}, <u>out</u> ReqProps = {WS : X, OS:UNIX})



Summary

Advantages of specifying with graph rewrite rules

- Graphs are an appropriate data model for a large set of applications.
- Even complex data structures with a high number of consistency constraints may be represented as graphs.
- Complex graph transformations may be specified declaratively by graph rewrite rules.
- Visual programs composed of graph rewrite rules are easy to comprehend.
- The specification is operational, code generation is supported (for rapid prototyping).

Disadvantages of specifying with graph rewrite rules

- Generality is constrained: commitment to a specific data model.
- For simple data types, graphs and graph transformations are an "overkill".
- Potential efficiency problems (subgraph search is NP-complete).
- In case of PROGRES:
  - Very expressive, but also very complex language.
  - Specifying-in-the-large not completely elaborated.

Literature

- G. Rozenberg (Ed.): **Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations**, World Scientific (1997)
  Collection of papers on the theory of graph rewriting systems
- H. Ehrig et al. (Eds.): **Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools**, World Scientific (1999)
  Applications of graph rewriting systems, including e.g. chapters on PROGRES and its applications
- A. Schürr: **Operationales Spezifizieren mit programmierten Graphersetzungssystemen**, Deutscher Universitätsverlag, Wiesbaden (1991)
  PROGRES language definition (in German)
- A. Schürr, B. Westfechtel: **Graphgrammatiken und Graphersetzungssysteme**, script for the corresponding lecture, Aachener Informatik-Berichte 92-15 (1992)
  Survey of the most important approaches concerned with graph rewriting