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

1 : List
   Name = "Months"

2 : Element
   Data = "January"

3 : Element
   Data = "February"

4 : Element
   Data = "April"

5 : Element
   Data = "May"
Example of a graph rewrite rule

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

\[ \begin{align*}
1' &= \text{list} \\
2' &= \text{elem} \\
3' &= \text{elem} \\
\end{align*} \]

\[ \text{Elem} \quad \text{Elem} \]

\[ \begin{align*}
1' &= 1 \\
2' &= 2 \\
4' &= \text{Element} \\
3' &= 3 \\
\end{align*} \]

\[ \text{Next} \quad \text{Next} \]

\[ \text{Attribute assignments} \]

\[ \text{Identically replaced node} \]

\[ \text{New node} \]

\[ \text{Input node} \]

\[ \text{Name} \]

\[ \text{Parameter} \]

\[ \text{Left-hand side} \]

\[ \text{Right-hand side} \]

\[ \text{return part} \]

\[ \begin{align*}
\text{attribute transfer} &\quad 4'.\text{Data} := \text{data}; \\
\text{return} &\quad \text{new} := 4'; \\
\text{end}; \\
\end{align*} \]
Application of the graph rewrite rule

1 : List
Name = "Months"

First  Elem  Last

2 : Element
Data = "January"

Next

3 : Element
Data = "February"

Next

4 : Element
Data = "April"

Next

5 : Element
Data = "May"

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

1 : List
Name = "Months"

First  Elem  Last

2 : Element
Data = "January"

Next

3 : Element
Data = "February"

Next

6 : Element
Data = "March"

Next

4 : Element
Data = "April"

Next

5 : Element
Data = "May"
Example of a graph test

```
test GetNextElement
   (list : List; elem : Element; out next : Element) =
   `1 = list
   Elem
   `2 = elem
   Next
   `3 : elem
   return next := `3;
   end;
```

Graph

Name

Parameter

Return part
Interface of the abstract data type **List**

<table>
<thead>
<tr>
<th>List creation and deletion</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateList</td>
<td>Creation of a list</td>
</tr>
<tr>
<td>DeleteList</td>
<td>Deletion of a list</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertFirstElement</td>
<td>Insertion into an empty list</td>
</tr>
<tr>
<td>PreInsertElement</td>
<td>Insertion before an element</td>
</tr>
<tr>
<td>PostInsertElement</td>
<td>Insertion after an element</td>
</tr>
<tr>
<td>DeleteElement</td>
<td>Deletion of an element</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Read operations</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IsEmpty</td>
<td>List empty?</td>
</tr>
<tr>
<td>GetFirstElement</td>
<td>First element</td>
</tr>
<tr>
<td>GetLastElement</td>
<td>Last element</td>
</tr>
<tr>
<td>GetNextElement</td>
<td>Next element</td>
</tr>
<tr>
<td>GetPreviousElement</td>
<td>Previous element</td>
</tr>
<tr>
<td>GetData</td>
<td>Data of current element</td>
</tr>
</tbody>
</table>
Cases of PreInsertElement

1: List
   Name = "Months"

2: Element
   Data = "January"
   PreInsertFirstElement

3: Element
   Data = "February"
   PreInsertSecondElement

4: Element
   Data = "April"
   PreInsertOtherElement

5: Element
   Data = "May"
   PreInsertLastElement
Example of programming with graph rewrite rules

Transaction

```
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) ⇔

- \( V \) 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

\[ G = (V, E, l) \] with:

- \( V = \{1, 2, 3, 4, 5\} \)
- \( E = \{(1, \text{First}, 2), (1, \text{Elem}, 3), (1, \text{Elem}, 4), (1, \text{Last}, 5), (2, \text{Next}, 3), (3, \text{Next}, 4), (4, \text{Next}, 5)\} \)
- \( l = \{(1, \text{List}), (2, \text{Element}), (3, \text{Element}), (4, \text{Element}), (5, \text{Element})\} \)
Partial graphs and subgraphs

Definition: Partial graph

\[ G = (V, E, l) \text{ is a partial graph of } G^\prime = (V^\prime, E^\prime, l^\prime) \iff \]
\[ V \subseteq V^\prime, \text{ i.e., the nodes of } G \text{ are also contained in } G^\prime. \]
\[ E \subseteq E^\prime, \text{ i.e., the edges of } G \text{ are contained in } G^\prime. \]
\[ l^\prime|_V = l, \text{ i.e., the nodes of } G \text{ have the same labels in } G^\prime. \]

Definition: Subgraph

\[ G = (V, E, l) \text{ is a subgraph of } G^\prime = (V^\prime, E^\prime, l^\prime) \iff \]
\[ G \text{ is a partial graph of } G^\prime. \]
\[ E = \{ (v_1, v_2) \in E^\prime | v_1, v_2 \in V \} \]
\[ G \text{ contains all edges of } G^\prime \text{ whose sources and targets are common to } G \text{ and } G^\prime. \]
Example of partial graphs and subgraphs

Graph

1 : List

2 : Element → Next → 3 : Element → Next → 4 : Element → Next → 5 : Element

Partial graph

1 : List

2 : Element → Next → 3 : Element → Next → 4 : Element

Subgraph

1 : List

4 : Element → Next → 5 : Element
Graph morphisms

Definition: Graph morphism
A function \( h : V \rightarrow V' \) is a graph morphism from \( G \) to \( G' \), i.e.,
\[
h : G \rightarrow G' \iff \\
\forall v \in V : l'(h(v)) = l(v), \text{ i.e., labels are preserved.} \\
\forall (v_1, el, v_2) \in E : (h(v_1), el, h(v_2)) \in E', \text{ i.e., edges are preserved.}
\]

Definition: Graph isomorphism
A graph morphism \( h : G \rightarrow G' \) is a graph isomorphism \( \iff \)
\[
\forall v \in V : l'(h(v)) = l(v), \text{ i.e., labels are preserved.} \\
\forall (v_1, el, v_2) \in E : (h(v_1), el, h(v_2)) \in E', \text{ i.e., edges are preserved.}
\]
\[
\iff h : V \rightarrow V' \text{ is injective and surjective} \\
\iff h : V \rightarrow V' \text{ induces a function } h' : E \rightarrow E', \text{ which must be injective and surjective, as well.}
\]
Definition: Union of graphs

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

$\Rightarrow G \cup G' = G'' = (V'', E'', I'')$ with

$\Rightarrow V'' = V \cup V'$

$\Rightarrow \forall v \in V'': I''(v) = \text{if } v \in V \text{ then } I(v) \text{ else } I'(v) \text{ end}$

$\Rightarrow E'' = E \cup E'$

$\Rightarrow G \oplus G'$ 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, $I \mid V \cap V = I' \mid V \cap V'$:

$G \setminus G' = G'' = (V'', E'', I'')$ with

- $V'' = V \setminus V'$
- $I'' = I \mid V'$
- $E'' = E \setminus E'$ (without deletion of dangling edges)

$G \setminus G' = G''' = (V''', E''', I''')$ with

- $V''' = V \setminus V'$
- $I''' = I \mid V'$
- $E''' = (E \setminus E') \cap (V'' \times L_e \times V'') = E \cap (V'' \times L_e \times V'')$
  (with deletion of dangling edges)
Set-theoretic graph operations (3)

Definition: Intersection of graphs

Let $G$ and $G'$ be directed graphs, $l | v \cap v = l' | v \cap v'$:

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

$V'' = V \cap V'$

$l'' = l | v''$

$E'' = E \cap E'$
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)

production DeleteElement =

 ::= 

end;
Example of a graph rewrite rule (2)

production DeleteElement -

```
1 : List
  Elem
  2 : Element
    Next
      3 : Element
        Next
          4 : Element
```

end;
Application of a graph rewrite rule

(Direct) derivation

A graph $G'$ is derivable from a graph $G$ by a rule $r = (L, K, R) ⇔$

» There is an isomorphism $h : L → G_L$, 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:

$$H = G \setminus (h(L) \setminus h(K))$$

» Nodes and edges of $R$ which do not belong to $K$ are inserted:

$$G' = H \oplus h'(R \setminus K),$$

where

$\Rightarrow h'$ maps nodes of $R \setminus K$ such that they do not occur in $H$,

$\Rightarrow$ 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 $\Rightarrow$
  - Higher flexibility by identification of graph elements
  - Danger of undesired effects of rule applications

- $G_L$ must even be a subgraph $G$ $\Rightarrow$
  - Larger left-hand sides
  - Larger rule sets

- No edges from deleted nodes to context nodes
  (Dangling Edge Condition) $\Rightarrow$
  - Exclusion of undesired side effects
  - Large left-hand sides

- Empty gluing graph $\Rightarrow$
  - 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 $R$ if $gs \iff$

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. $n → n + 1$: 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 = PROgrammed GRaph REwriting Systems
- 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

- **PROGRES language**
  - written in
  - defines
  - specifies
  - analyzes and executes
  - implements

- **PROGRES specification**
  - defines
  - implements
  - written in

- **PROGRES environment**
  - generates
  - defines

- **Generated tools**
  - generates
Example

Tools for programming-in-the-large

```
Example

system Example

Main

Variant V1:
WS: ?
OS: ?

Body variant

variant uses

ado

module uses

UserInterface

Variant V1:
WS: X
OS: UNIX

Body import

Interface import

variant uses

Module interface

Module body

Files

Variant V2:
WS: ?
OS: MSDOS

Variant V1:
WS: ?
OS: UNIX

import

import
```
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 \( \text{WS} = X \) and \( \text{OS} = \text{UNIX} \).)
Attributed Graphs and Graph Schemata
Representation of the example as an attributed graph

Node (no. + type)

1 : System
  Name = "Example"
  Size = 11600
contains

2 : FunctionModule
  Name = "Main"
  Size = 2100
contains

3 : Variant
  Name = "V1"
  Size = 2000
  Props = {}
contains
4 : ADOModule
  Name = "UserInterface"
  Size = 6000
contains
5 : Variant
  Name = "V1"
  Size = 5000
  Props = {WS:X, OS:UNIX}
contains

6 : ADTModule
  Name = "Files"
  Size = 3500
contains

7 : Variant
  Name = "V2"
  Size = 2000
  Props = {OS:MSDOS}
contains
8 : Variant
  Name = "V1"
  Size = 3000
  Props = {OS:UNIX}
contains

9 : Config
  Name = "C"
  Size = 10000
  Props = {WS:X, OS:UNIX}
contains

Attribute (name, value)

contains
has

has

contains
m_uses

v_uses

v_uses
Graphical definition of a graph schema

- **Node type**
  - Variant
  - System
  - Config

- **Instance of relationship**
  - Variant
  - System
  - Config

- **Edge type**
  - v_uses
  - m_uses

- **Specialization**
  - has
  - contains

- **Node class**
  - MODULE
  - VARIANT
  - SYSTEM
  - CONFIGURATION

- **Graphical definition of a graph schema**
  - ADTModule
  - FunctionModule
  - ADOModule

- **REALIZATION**
  - SPECIFICATION
  - COMPLEX

- **UNIT**

- **Instance of relationship**
  - v_uses
  - m_uses
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)

```plaintext
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 *)
```

(* A 'REALIZATION' is an implementation of a 'SPEC.' *)

(* A 'REALIZATION' is an implementation of a 'SPEC.' *)
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'. *)
Textual definition of a graph schema (3)

```latex
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):
    \[ v_1 = p => v_2 \iff \text{There is a path } p \text{ from } v_1 \text{ to } v_2 \]
  - a function on node sets:
    \[ p(V) = \{ v_2 | \exists v_1 \in V : v_1 = p => v_2 \} \]

- 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p &amp; q$</td>
<td>Concatenation of $p$ and $q$</td>
</tr>
<tr>
<td>$p \text{ or } q$</td>
<td>Connection by $p$ or $q$</td>
</tr>
<tr>
<td>$p \text{ and } q$</td>
<td>Connection by $p$ and $q$</td>
</tr>
<tr>
<td>$p \text{ but not } q$</td>
<td>Connection by $p$ but not by $q$</td>
</tr>
<tr>
<td>$[p \mid q]$</td>
<td>Connection by $p$, else by $q$</td>
</tr>
<tr>
<td>$p^+$</td>
<td>Transitive closure</td>
</tr>
<tr>
<td>$p^*$</td>
<td>Reflexive and transitive closure</td>
</tr>
<tr>
<td>${p}$</td>
<td>Maximal iteration</td>
</tr>
</tbody>
</table>
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

- **Single node**
  - Node 1: NodeType1
  - Node 2: NodeType2

- **Edge**
  - Node 3: NodeType3
  - Node 4: NodeType4

- **Set node**
  - Node 5: NodeType5
  - Node 6: NodeType6
  - Path 1

- **Path**
  - Node 7: NodeType7
  - Node 8: NodeType8

- **Optional node**
  - Node 9: NodeType9
  - Node 10: NodeType10

- **Negative edge**
  - Node 4: NodeType4

- **Optional set**
  - Node 6: NodeType6

- **Restriction**
  - Node 3: NodeType3

- **Negative path**
  - Node 7: NodeType7

- **Negative node**
  - Node 9: NodeType9
Example for parameters, negative nodes, attribute transfer, etc.

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

`1 : System
contains
`2 : MODULE

name(MName)

 ::= 

1' = '1
contains
3' : MType

(* 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'. *)
Example of optional set nodes

production DeleteModule(Module : MODULE) =

`1 = Module

has

`2 : VARIANT

::=

end;
Example of a negative path

production CreateMUse(Client, Server : MODULE) =

'1 = Client

\textbf{DependsOn} \quad '2 = Server

::=

\begin{align*}
  1' &= '1 \\
  m\textunderscore uses &\rightarrow \\
  2' &= '2
\end{align*}

end;

(* Creates a new import from ‘Client’ to ‘Server’. *)
**Embedding rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy <code>e</code>→ from <code>n</code> to <code>m</code></td>
<td>Copy outgoing <code>e</code> edges from node <code>n</code> of the left-hand side to node <code>m</code> of the right-hand side</td>
</tr>
<tr>
<td>remove <code>e</code>→ from <code>n</code></td>
<td>Delete outgoing <code>e</code> edges from node <code>n</code></td>
</tr>
<tr>
<td>redirect <code>e</code>→ from <code>n</code> to <code>m</code></td>
<td>Redirect outgoing <code>e</code> edges from <code>n</code> to <code>m</code></td>
</tr>
<tr>
<td>&lt;-<code>e</code> instead of <code>e</code>→</td>
<td>Analogously for incoming rather than outgoing edges</td>
</tr>
<tr>
<td><code>-e</code>→ as <code>-f</code>→ instead of <code>e</code>→</td>
<td>Relabeling</td>
</tr>
<tr>
<td><code>-e</code>→ as <code>&lt;-f</code> statt <code>e</code>→</td>
<td>... and change of direction</td>
</tr>
</tbody>
</table>
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

production ResolveImport( C : CONFIGURATION ; ReqProps : string [3:n] ;
out NewProps : string [0:n])

rule editor

needs
contains
has
contains
has

4' = '4
5' = '5
contains
contains
has

condition '2'.Props 'are_in' ReqProps;
return NewProps := merge ( ReqProps, '2'.Props );
end;
("Selects a module which is used by a 'Variant' in ")
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):
  - **Atomic**: either complete execution or no modification of the host graph
  - **Consistent**: consistency-preserving transformation
  - **Isolated**: isolation in multi-user mode
  - **Durable**: persistent
- Additional property: **non-determinism**
- Failure of executing an operation results in **backtracking**
## Overview of control structures

<table>
<thead>
<tr>
<th>Control structure</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p &amp; q)</td>
<td>Sequence</td>
</tr>
<tr>
<td>(p \text{ and } q)</td>
<td>(p \text{ und } q) in any order</td>
</tr>
<tr>
<td>(p \text{ or } q)</td>
<td>Non-deterministic choice</td>
</tr>
<tr>
<td>\textbf{choose} (p_1) \textbf{else} (p_2) ... \textbf{end}</td>
<td>Try (p_1), else (p_2) ...</td>
</tr>
<tr>
<td>\textbf{loop} (p) \textbf{end}</td>
<td>Loop (execute (p) as long as possible)</td>
</tr>
<tr>
<td>\textbf{for} all (n) \textbf{do} (p) \textbf{end}</td>
<td>Execute (p) for all nodes (n)</td>
</tr>
<tr>
<td>\textbf{use} (v : \ldots) \textbf{do} (p) \textbf{end}</td>
<td>Block with declaration of variables</td>
</tr>
</tbody>
</table>
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) =

`1 : System contains `2 : MODULE isMain

`3 : VARIANT

has

:=

1′ = `1 contains 2′ = `2

has

has

4′ : Config contains 3′ = `3

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

4: ADOModule
Name = "UserInterface"
has

5: Variant
Name = "V1"
Props = {WS:X, OS:UNIX}
v_uses

6: ADTModule
Name = "Files"
m_uses

7: Variant
Name = "V2"
Props = {OS:MSDOS}
v_uses

8: Variant
Name = "V1"
Props = {OS:UNIX}
Example (2)

```
InitConfig("C", {}, out ReqProps = {})
```

1 : System
Name = "Example"
contains

2 : FunctionModule
Name = "Main"
has

3 : Variant
Name = "V1"
Props = {}
contains

4 : ADOModule
Name = "UserInterface"
v_uses

5 : Variant
Name = "V1"
Props = {WS:X, OS:UNIX}
v_uses

6 : ADTModule
Name = "Files"
has

7 : Variant
Name = "V2"
Props = {OS:MSDOS}
m_uses

8 : Variant
Name = "V1"
Props = {OS:UNIX}
has

9 : Config
Name = "C"
contains

```
Example (3)

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

1 : System
Name = "Example"
contains

2 : FunctionModule
Name = "Main"
has

3 : Variant
Name = "V1"
Props = {}
has

4 : ADOModule
Name = "UserInterface"
m_uses

5 : Variant
Name = "V1"
Props = {WS:X, OS:UNIX}
v_uses

6 : ADTModule
Name = "Files"
m_uses

7 : Variant
Name = "V2"
Props = {OS:MSDOS}
v_uses

8 : Variant
Name = "V1"
Props = {OS:UNIX}
v_uses

9 : Config
Name = "C"
contains
Example (4)

- ResolveImport fails for the module UserInterface because \{OS:UNIX\} in \{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", {}, out ReqProps = {OS:UNIX})
Example (6)

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

1 : System
Name = "Example"
contains

2 : FunctionModule
Name = "Main"
has
contains

3 : Variant
Name = "V1"
Props = {}
contains

4 : ADOModule
Name = "UserInterface"
has
contains

5 : Variant
Name = "V1"
Props = {WS:X, OS:UNIX}

6 : ADTModule
Name = "Files"
contains

7 : Variant
Name = "V2"
Props = {OS:MSDOS}

8 : Variant
Name = "V1"
Props = {OS:UNIX}

9 : Config
Name = "C"
contains
contains
contains

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

  *Collection of papers on the theory of graph rewriting systems*

  *Applications of graph rewriting systems, including e.g. chapters on PROGRES and its applications*

  *PROGRES language definition (in German)*

  *Survey of the most important approaches concerned with graph rewriting*