RWTH Aachen

Universität Bw München

The PROGRES Language Manual Version 9.x

The PROGRES Developer Team

Lehrstuhl für Informatik III
RWTH Aachen

Institut für Softwaretechnologie
Universität der Bundeswehr München

Table of Contents

The PROGRES Language Manual (Version 9.x)

1 Introduction

The PROGRES language definition (version 9.x) is a mixture of Extended Backus Naur Form (EBNF) rules and natural language texts. EBNF rules are used to define the language's context-free syntax; the running text explains the semantics of each introduced construct briefly. For a complete formal definition of an elder version of PROGRES the reader is referred to [Sch91] and to [Sch97]. Further information about the language PROGRES in various projects are published in [Wes91, AE94, SWZ95b, SWZ95a, SWZ96, Zün96, Nag96, KS97, HKW98]

The definition of PROGRES' syntax by means of EBNF rules is possible due to the fact that any graphical language element has a textual "Ersatz"-representation and that the translation from the textual representation to its graphical counterpart is rather trivial. A discussion of other less well-known syntax definition formalisms, which are especially tailored towards the definition of purely visual languages, may be found in [RS97].

The first part of this appendix is a definition of the *EBNF definition language* itself. It uses the same mixture of EBNF rules for context-free syntax definition purposes and natural language comments for additional explanations.

Please note that this paper is a  *ertext* document. It is available in three different formats:

(1) The Framemaker document processing system hypertext (view) format as part of the PROGRES release: $PROGRESROOT/man/ProgresSyntax.fm.
The layout of this file is identical with the layout of the printed appendix version.

(2) The *pdf* format processed by Adobe's acrobat reader as part of the PROGRES release: $PROGRESROOT/man/ProgresSyntax.pdf.
The layout of this file is almost identical with the layout of the printed appendix version and it has a tree-like index structure.

(3) The WWW format *html* for any available internet browser, accessible via the URL: http://www-i3.informatik.rwth-aachen.de/research/progres/ProgresSyntax.html.

Any applied occurrence of a nonterminal on the following pages is a "clickable" link to its definition, although these occurrences are *not underlined*. Underlined keywords on the other hand are just keywords and not links to different parts of the document.

Furthermore, there are two types of nonterminals on the right-hand side of EBNF rules, which are treated differently: nonterminals starting with prefix "Decl" or "Appl". They denote identifier declarations or applied identifier occurrences. The default lexical syntax of these identifiers is a sequence of alphanumerical characters. Any deviation from this law is explained in the running text. For further details concerning this topic the reader is referred to the language's lexical syntax definition in (f)lex input format, which is part of the programming environment source code.

2 Context-free Syntax of the Syntax Definition Language EBNF

(1) **EBNF** ::=
`{ EBNF_Rule } EBNF_Rule`

A context-free syntax definition is a nonempty list of EBNF rules.

(2) **EBNF_Rule** ::=
`DeclNonterminalId "::=" EBNF_Expression`

Each EBNF rule introduces one class of nonterminals. Its name (identifier) precedes the separator "`::=`", its definition follows the separator.

(3) **DeclNonterminalId** ::= ...

This is the declaration of a nonterminal identifier. It has to be a sequence of alphanumeric characters (including "-" and "_" as separators).

(4) **EBNF_Expression** ::=
`Sequence`
`| Choice`
`| SimpleExpression`

An EBNF expression is either a complex expression or a basic expression.

(5) **Sequence** ::=
`{ SimpleExpression } SimpleExpression`

A sequence is a nonempty list of EBNF subexpressions. These subexpressions are usually applied occurrences of nonterminal identifiers. The right-hand side of the `Sequence` rule itself is e.g. a sequence of two EBNF subexpressions, where the first one is a `List` subexpression and the second one an `ApplNonterminalId`.

(6) **Choice** ::=
`{ SimpleExpression "|" } SimpleExpression`

A choice is an EBNF subexpression with a nonempty list of alternatives.

(7) **SimpleExpression** ::=
`Keyword | TerminalId | ApplNonterminalId`
`| Optional | List`

A simple expression is either a keyword or a (non-)terminal identifier or a more complex expression enclosed by delimiters.

(8) **Keyword** ::= ...

A keyword is either an underlined sequence of letters (plus the separator "_" which looks like an underlined blank) or an arbitrary sequence of characters enclosed by "‴" as the leading symbol and "‴" as the trailing symbol. Examples of legal keywords are: `begin` or `is a` or `"->"`, but not `*` or `1245`.

(9) **TerminalId** ::= ...

The definition of the lexical syntax of identifiers and constants is outside the scope of our EBNF rules. They are either sequences of alphanumeric characters (including "_" as separator) with a letter as the leading character or explained in more detail in the running text.

(10) **ApplNonterminalId** ::= ...

This is an applied occurrence of a nonterminal identifier, defined elsewhere in the EBNF. Recursion is allowed, but rarely used due to the existence of a special list construction operator.

(11) **Optional** ::=
`"[" EBNF_Expression "]"`

An optional EBNF expression is an EBNF expression enclosed by square brackets. It has the usual meaning, i.e. denotes an optional construct of the defined language.

(12) **List** ::=
`"{" EBNF_Expression "}"`

A list is an EBNF subexpression enclosed by curly braces. It is used to define lists of 0 to n elements, where each element belongs to the sublanguage defined by the given subexpression. The right-hand side of the `List` rule is a sequence of three simple EBNF subexpressions (two keywords and one nonterminal identifier).

3 Context-Free Syntax of the Language PROGRES

The following pages describe the syntax and semantics of the language PROGRES (version 9.x) as precisely as possible. Nevertheless, there are still some differences between the syntax presented here and the syntax of the released language implementation (version 9.2), which is documented in the file $PROGRESROOT/man/ProgresSyntax:

- Repair actions of local constraints (cf. rules on page 10) are valid for a list of constraint declarations and not just for a single declaration.
- Pre- and postconditions of productions (cf. rules on page 14) are not mentioned in this document although being supported.
- Sets may not yet be constructed using curly braces as in `{1,2,3}`. They have to be defined using the union operator as in `1 or 2 or 3`.

(1) **Specification** ::=
  spec[illegible]clSpecId
    [OptImportList]
    [OptDeclList]
  end "."

A specification has a name and contains a list of external type and function imports as well as a list of declarations of node types, edge types, productions, etc.

(2) **OptDeclList** ::=
  { Declaration } Declaration

This is a nonempty list of declarations. Each declared item has its own unique identifier with global visibility. The order of declarations in the declaration list is of no importance.

(3) **Declaration** ::=
    Section
  | NodeClassDecl (cf. rules on page 5)
  | NodeTypeDecl (cf. rules on page 5)
  | EdgeTypeDecl (cf. rules on page 5)
  | ConstraintDecl (cf. rules on page 11)
  | QueryDecl (cf. rules on page 13)
  | TransactionDecl (cf. rules on page 13)
  | TestDecl (cf. rules on page 20)
  | ProductionDecl (cf. rules on page 20)
  | PathDecl (cf. rules on page 28)
  | RestrictionDecl (cf. rules on page 29)
  | FunctionDecl (cf. rules on page 35)

This is the full list of all possible types of declarations.

(4) **Section** :
  section[illegible]clSectionId
    [OptImportList]
    [OptDeclList]
  end ";"

Sections are a very primitive means to structure a large specification. They do not build their own name scopes, i.e. it does not matter at all whether a certain import or declaration is inside a certain section or not. They will be replaced soon by proper modules (packages). One exception from this rule concerns the treatment of defined global integrity constraints (cf. rules on page 11). They affect the behavior of those graph transformations only, wich are part of the same section or its subsections.

3.1 Import of External C Types and Functions

(5) **OptImportList** ::=
```
{ ModuleImport }
  ModuleImport
```
A specification imports attribute types and functions from a host programming language, which is usually C. The import is restricted to non-pointer types of fixed size or varying size (with known upper boundary). For further details see documentation of PROGRES release (file `$PROGRES-ROOT/import/README`).

(6) **ModuleImport** ::=
```
from DeclModuleId import
  { Import }
    Import end ";"
```
Import a number of resources from the host programming language module (file) `DeclModuleId`. These resources are linked at run-time to the importing specification. The order of list elements is of no importance.

(7) **Import** ::=
```
  TypeImportList
| FunctionImportList
```
An import is either a list of attribute types or a list of functions over these attribute types.

(8) **TypeImportList** ::=
```
types
  { DeclAttTypeId "," }
    DeclAttTypeId ";"
```
This is the list of imported attribute type identifiers, which have to be defined in the designated host programming language module. The order of list elements is of no importance.

(9) **FunctionImportList** ::=
```
functions
  { FunctionImportDecl "," }
    FunctionImportDecl ";"
```
This is the list of imported functions (together with their parameter profiles), which have to be defined in the designated host programming language module. The order of list elements is of no importance.

(10) **FunctionImportDecl** ::=
```
FuncOpName ":" FunctionType
```
An imported function is either a normal function with an arbitrary number of in-parameters or an infix operator with two in-parameters or a prefix operator with one in-parameter.

(11) **FunctionType** ::=
```
[ OptTypeList ] "->" Type
```
An imported function's parameter profile is an optional list of in-parameter types followed by the function's always existing return-value.

(12) **OptTypeList** ::=
```
"(" { Type "," } Type ")"
```
The in-parameter type list is just a sequence of type definitions separated by ",". The order of type definitions in this list reflects the function's order of formal in-parameters.

3.2 Node Classes, Node Types, and Edge Types

(13) **NodeClassDecl** ::=
```
node class DeclNodeClassId [ OptSuperClassIdList ] [ ";" ]
  [ OptAttDeclList ]
  [ OptRuleList ]
end ";"
```
A node class has a maybe empty list of direct superclasses, an optional list of attribute (and constraint) declarations, and an optional list of attribute rule redefinitions. The extension of a node class is not a set of nodes but a set of node types, which have this class as direct or indirect superclass. Node classes are always abstract, i.e. they do not have any direct node instances. The often used term

"n is a node of node class `C`"

is, therefore, a shorthand for

"n is an instance of a node type which belongs directly or indirectly to node class `C`".

The graphical representation of a node class declaration is a box, which has the node class name as inscription (plus its attribute declarations if desired).

(14) **OptSuperClassIdList** ::=
```
is a { ApplNodeClassId "," } ApplNodeClassId
```
The list of direct superclasses of a class. A class inherits all properties of its superclasses. The order of list elements is not important. The empty list is used for node classes without any superclasses. Any two classes may not have more than one smallest common superclass and more than one greatest common subclass, i.e. the class hierarchy has to be a lattice, after a greatest and a smallest class are added.

The graphical representation of the `is a` relationship is a dashed arrow from the subclass to the superclass.

(15) **NodeTypeDecl** ::=
```
node type DeclNodeTypeId : ApplNodeClassId [ ";" ]
  [ OptAttDeclList ]
  [ OptRuleList ]
end ";"
```
A node type belongs to one and only one class. It inherits all properties of its class. Any node instance belongs to one and only one node type, which determines its behavior (concerning attribute evaluation rules and constraints). Node types are always final, i.e. they are the leaves of the node class hierarchy. For this reason their extensions are nonoverlapping sets of node instances.

The graphical representation of a node type declaration is a box with round corners, which has the node type name as inscription (plus its attribute declarations if desired). This box should always be the source of a (single) dashed arrow, which has the node type's (super-)class as target.

(16) **EdgeTypeDecl** ::=
```
edge type DeclEdgeTypeId ":" RelType;
```
An edge type is a binary, directed but nevertheless bidirectional relation between nodes of certain classes or types. Edges may be traversed in both directions from source node to target node and from target node to source node. Furthermore, referential integrity is guaranteed by the fact that any node deletion operation has the the side-effect to delete all adjacent edges, too.

The graphical representation of an edge type declaration is a solid arrow from the source to the target node type or class. It has the edge type name as its label.

(17) **RelType** ::=
```
ApplTypeClassId [ Qualifier ] "->" ApplTypeClassId [ Qualifier ]
```
Defines source and target node types or classes as well as multiplicity of binary relationships between nodes of these types or classes. The example

```
T [l_1:u_1] -> C [l_2:u_2]
```

defines for instance a directed relationship from nodes of type `T` to nodes of a type of class `C`. Any node of type `T` is related to at least `l_2` and at most `u_2` nodes of class `C`. A node of class `C` is related to at `l_2` and at most `u_2` nodes of type `T`. The default for omitted optional qualifiers is `[0:n]` (cf. rules on page 46 for the definition of qualifiers).

3.3 Attribute Declarations and Redefinitions

(18) **OptAttDeclList** ::=
```
{ AttDecl } AttDecl
```
List of attribute declarations and local constraints of node class or node type. The order of list elements is insignificant.

(19) **AttDecl** ::=
```
  IAttDeclList
| DAttDeclList
| MAttDeclList
| ConstrDeclList
```
There are three categories of attributes plus local constraints. The attribute declaration syntax with lists in lists allows one to use the same keyword "`intrinsic`" or ... for a number of related attributes of the same category.

(20) **IAttDeclList** ::=
```
intrinsic { IAttDecl } IAttDecl
```
List of related intrinsic (normal) node attributes. The order of list elements is of no importance.

(21) **DAttDeclList** ::=
```
derived { DAttDecl } DAttDecl
```
List of related derived node attribute declarations. The order of list elements is of no importance.

(22) **MAttDeclList** ::=
```
meta { MAttDecl } MAttDecl
```
List of related meta node type attribute declarations. The order of list elements is of no importance.

(23) **ConstrDeclList** ::=
```
constraint { ConstrAttDecl } ConstrAttDecl
```
List of related local constraint declarations. The order of list elements is of no importance.

(24) **OptRuleList** ::=
```
{ Rule } Rule
```
List of attribute redefinitions or constraint redefinitions. Any subclass or type may redefine the (initial) value defining expressions of its inherited attributes and constraints. Multiple inheritance conflicts (a class inherits two different expressions for the same attribute or constraint) from two different superclasses are recognized and must be resolved by redefinition.

(25) **OptRule** ::=
```
  IAttRuleList
| DAttRuleList
| MAttRuleList
| ConstrRuleList
```
For any category of attributes (including local constraints) there is one category of attribute redefinition rule list. Intrinsic attributes are redefined using an `IAttrRuleList`, etc. Such a redefinition changes the associated expression of an attribute, but not its type.

(26) **IAttRuleList** ::=
```
redef intrinsic { IAttRule } IAttRule
```
List of related intrinsic node attribute redefinitions. The order of list elements is of no importance.

(27) **DAttRuleList** ::=
```
redef derived { DAttRule } DAttRule
```
List of related derived node attribute redefinitions. The order of list elements is of no importance.

(28) **MAttRuleList** ::=
```
redef meta { MAttRule } MAttRule
```
List of related meta node type attribute redefinitions. The order of list elements is of no importance.

(29) **ConstrRuleList** ::=
```
redef constraint { ConstrAttRule } ConstrAttRule
```
List of related local constraint redefinitions. The order of list elements is of no importance.

3.4 Intrinsic Attribute Declarations and Redefinitions

(30) **IAttDecl** ::=
[OptKeyOrIndex] ApplIAttId [“:” Type]
[“:=” Expression] “;”

An intrinsic attribute has a name (identifier) plus a type plus an optional constant initial value. The initial value computing expression may contain references to meta attributes of the same node (type), but not to intrinsic or derived attributes of the same or different nodes in a graph.

Any intrinsic attribute read access, which is not preceded by a write access, leads to the evaluation of the appropriate initial value defining expression. The default expression is `halt`, which stops the execution process immediately.

The type definition may be omitted if it can be inferred from the attribute’s initial value defining expression. The declaration

```
intrinsic Names := "Balu" or "Kaa"
```

is equivalent to

```
intrinsic Names : string [1:n] := "Balu" or "Kaa" .
```

An intrinsic attribute’s type may be any type which is a legal type for variables, parameters, etc. But note that attribute declarations with node types or node classes as types are not interpreted as unidirectional pointers to other nodes, but as a short-hand for a normal edge type declaration. Let us assume that a node type or node class `NODE` possesses the following attribute declarations:

```
intrinsic key Contains: ELEMENT [1:n];
          References: ELEMENT [0:n];
```

They are translated as follows into edge type declarations

```
edge type Contains: NODE [0:1] -> ELEMENT [1:n];
edge type References: NODE [0:n] -> ELEMENT [0:n];
```

The cardinality constraint for the target node is derived from the attribute type’s cardinality constraint, the cardinality constraint for the source node is either the default `[0:n]` or `[0:1]` if the attribute is declared as a key attribute (for each attribute value = target node exists at most one source node). The normal usage of the keywords `key` and `index` is explained below.

(31) **IAttRule** ::=
ApplIAttId “:=” Expression “;”

Redefinition of the initial value defining expression for a given intrinsic attribute in a node class `C` or a node type `T`. This attribute must be an inherited attribute of the node class `C` or the node type `T`. The new expression overrides the inherited expression and may be overridden again. Dynamic binding is used to determine at run-time which expression has to be evaluated to compute the initial attribute value of the regarded node (any node has a unique node type which determines its attribute evaluation functions).

Initial intrinsic attribute value definitions may be omitted if all node creating productions assign immediately appropriate values to the attributes of all new nodes.

Multiple inheritance conflicts have to be resolved manually by redefining the expression for the inherited attribute in the affected class. A multiple inheritance conflict occurs if one class inherits two different evaluation rules from two uncomparable superclasses `A` and `B` (neither `A` is a subclass of `B` nor `B` is a subclass of `A`). The smallest example of a node class hierarchy with a multiple inheritance conflict looks like follows:

```
node class S;
  intrinsic a := 1;
end;
node class A is a S;
  redef intrinsic a := 3;
end;
node class B is a S;
  redef intrinsic a := 4;
end;
node class Conflict_Class is a A, B end;
```

3.5 Derived Attribute Declarations and Redefinitions

(32) **DAttDecl** ::=
```
[ OptKeyOrIndex ] ApplIAttId [ ":" Type ]
                             [ "=" Expression ] ";"
```

A derived attribute has a name (identifier) plus a type plus a directed equation which contains references to other attributes of the same node or different nodes. These references are defined as path expressions which start at the regarded node `self` and traverse an unlimited number of edges in the general case. The evaluation of a derived attribute results in a run-time error if it depends directly or indirectly on its own value.

There is an important difference between the initial value definition of intrinsic attributes and the directed equation of a derived attribute. The first one is evaluated once after the creation of a new node attribute. The latter one will be reevaluated whenever one of the referenced attributes in the directed equation changes its value.

The `Ancestors` attribute below is one example of a derived attribute. It has the union of the name attribute values of all those nodes as value, which are reachable from the current node `self` by traversing `parent` edges only:

```
derived Ancestors = self.(-parent-> +).Name
```

The attribute type definition may be omitted if it can be inferred from the attribute's directed equation, as it is the case for the attribute `Ancestors`. Its type is `string [0:n]` and not `string [1:n]` due to the fact that a graph may contain nodes without any outgoing `parent` edges.

A derived attribute's type may be any type which is a legal type for variables, parameters, etc. But note that attribute declarations with node types or node classes as types are not interpreted as unidirectional pointers to other nodes, but as a short-hand for a normal path declaration. Let us assume that a node type or node class `PERSON` contains the following attribute declaration:

```
derived ancestors: PERSON [0:n] = self.-parent-> +;
```

It is translated as follows into a path declaration:

```
static path ancestors: PERSON [0:n] -> PERSON [0:n] =
   -parent-> +
end;
```

The cardinality constraint for the target node is derived from the attribute type's cardinality constraint, the cardinality constraint for the source node is either the default `[0:n]` or `[0:1]` if the attribute is declared as a key attribute.

Defining a derived binary relationship not as a path but as a derived attribute has two important consequences: (1) Subclasses or types of the class `PERSON` may redefine the original definition of the derived relationship (the expression of the declaration). (2) The derived relationship is materialized and thereby equivalent to a so-called `static` path declaration (its evaluation leads to the creation of an appropriate edge which may then be traversed in both directions without any needs to reevaluate the path again).

(33) **DAttRule** ::=
```
ApplDAttId "=" Expression ";"
```

Redefinition of the directed equation for a given derived attribute in a node class `C` or a node type `T`. This attribute must be an inherited attribute of the node class `C` or the node type `T`. The new expression overrides the inherited expression and may be overridden again. Dynamic binding is used to determine at run-time which expression determines the derived attribute's value for the regarded node (any node has a unique node type which determines its attribute evaluation functions).

Abstract node classes need not possess evaluation functions for their derived attributes, but any concrete node type must possess an evaluation function for any (inherited) derived attribute. The default value for undefined evaluation functions is `halt` , which stops the execution process immediately.

Multiple inheritance conflicts are handled in the same way as in the case of intrinsic attributes.

3.6 Meta Attribute Declarations and Redefinitions

(34) **MAttDecl** ::=

```
DeclMAttId  [ ":" Type ]
            [ ":=" Expression ] ";"
```

A meta attribute has a name (identifier) plus a type plus a constant value. The value computing expression may contain references to meta attributes of the same node (type), but not to intrinsic or derived attributes of the same or different nodes in a graph. A meta attribute is not a node attribute, but a node type attribute. Its value may not be changed at run-time.

Meta attributes are useful for simulating genericity and for allowing a specification to inspect parts of its own graph type definition. The node class declaration

```
node class CONTAINER;
  meta ElementType : type in ELEMENT;
  intrinsic contains: ELEMENT [0:n];
  constraint self.contains.type = self.type.ElementType;
end;
```

introduces a `CONTAINER` class, which contains a set of nodes of type `ElementType`, which must be derived from node class `ELEMENT`. A specific `CONTAINER` type may be defined as follows:

```
node type FoodBin : CONTAINER;
  redef meta ElementType := Food;
end;
node type Food : ELEMENT end;
```

Please note that the current PROGRES version checks the constraint that a `FoodBin` contains `Food` nodes only at run-time and not at compile-time.

(35) **MAttRule** ::=

```
ApplMAttId ":=" Expression ";"
```

Redefinition of the constant value for a given meta attribute in a node class `C` or a node type `T`. This attribute must be an inherited attribute of the node class `C` or the node type `T`. The new expression overrides the inherited expression and may be overridden again. Dynamic binding is used to determine at run-time which expression determines the meta attribute's value for the regarded node type.

Abstract node classes need not possess evaluation functions for their derived attributes, but any concrete node type must possess an evaluation function for any (inherited) derived attribute. The default value for undefined evaluation functions is `halt` , which stops the execution process immediately.

Multiple inheritance conflicts are handled in the same way as in the case of intrinsic attributes.

(36) **OptKeyOrIndex** ::= Key | Index

These keywords usually enforce the creation of an attribute index that supports efficient associative access to all nodes with a given attribute value. They may be used together with intrinsic or derived attributes, which have a standard type or an imported type definition with default cardinality constraint `[1:1]` and a maximum byte length less equal than 250 (as e.g. the standard type `string`). Until now the PROGRES compiler uses attribute indexes for attribute conditions of the following form :

```
condition Node.IndexAttr = Expression;
```

(37) **Key** ::= key

Defines an intrinsic or derived attribute which has a unique value (such that two different nodes never have the same attribute value). Furthermore, the keyword enforces the creation of an attribute index. Please note that the keyword `key` has a rather different meaning if applied to an attribute, which has a node type or node class as its type definition (cf. `IAttDecl` rule above).

(38) **Index** ::= index

Enforces the creation of an attribute index. It has no impact onto the semantics of a given specification, but may increase the efficiency of certain graph transformations considerably.

3.7 Local Node Constraint Declarations and Redefinitions

(39) **ConstrAttDecl** ::=
```
DeclConstrAttId [ "=" Expression ]
                [ OptAttRepairAction ] ";"
```

A local constraint declaration introduces an integrity constraint, which is an arbitrary first-order predicate logic formula and which is checked at run-time. The constraint defining `boolean` expression uses path expressions to navigate from the regarded node `self` to related nodes in its neighborhood. Constraints are, therefore, a kind of derived `boolean` attributes, which have to be `true` at the end of so-called `safe` transactions or productions (cf. rules on page 13 and rules on page 20). The violation of a constraint causes immediate termination of a running execution process or the activation of a so-called repair action (the default repair action is `halt`).

The constraint declaration (without any repair action)

```
constraint ACyclicAggregation = not (self in self.-contains-> +);
```

requires, for instance, that a node may not contain itself, i.e. that `contains` edges do not form cycles in a graph. The introduced constraint identifier `ACyclicAggregation` allows one to refine the inherited constraint for certain subclasses or node types.

(40) **ConstrAttRule** ::=
```
ApplConstrAttId "=" Expression
                  [ OptAttRepairAction ] ";"
```

Redefinition of an inherited constraint for a certain subclass or type of nodes. The refining expression should be a logical consequence of the overridden expression, but this property is not checked (neither at compile-time nor at run-time). The redefinition may also redefine the inherited constraint's repair action as needed.

(41) **OptAttRepairAction** ::=
```
else
  StatExpr
end
```

This is the repair action of a local constraint declaration or redefinition. Any constraint with such a repair action is termed "active constraint" in contrast to a "passive constraint", which has `halt` as its default repair action. The detection of a violated constraint triggers the execution of its associated repair action. The execution of this action should not fail and repair the violated constraint. It may cause violation of other constraints and thereby trigger the execution of their repair actions. Execution stops if the default repair action `halt` is called or the called user-defined action is not able to remove the detected inconsistency.

The example below triggers the deletion of a `contains` edge between a (unique) parent node and its child node `self` if the parent node is a direct or indirect child of its own child node.

```
constraint ACyclicAggregation =
  not (self in self.-contains-> +);
else
  use Parent := self.<-contains- do
    DeleteContainsRelation(Parent, self)
  end
end;
```

Multiple inheritance conflicts are handled in the same way as in the case of intrinsic attributes.

(42) **OptRepairAction** ::=
```
else
  StatExpr
```

This is the repair action of a global graph constraints (see next rule). The only difference between the repair action of a local constraint and the repair action of a global constraint is the position of the keyword `end`.

3.8 Global Graph Constraint Declarations

(43) **ConstraintDecl** ::=
`constraint` ConstraintBody `end` ";"

A global constraint requires or forbids the existence of certain graph patterns. As a general rule, a global constraint should be used for those static integrity constraints, which regard the attributes or context of more than one node instance. On the contrary, a local node class or type constraint (cf. rules on page 10) should be used, whenever the regarded integrity constraint restricts the attribute values and the context of a single node instance. It is often a matter of taste whether a regarded integrity constraint deals with two (or more) node instances, which have the same importance, or whether it concerns mainly one node instance, which has the other inspected node instance as context (please note that the constraint checker for global constraints is a rather inefficient brute force algorithm, whereas the constraint checker for local constraints is a variant of the incremental derived attribute evaluation algorithm).

A global constraint is checked as the last action of all `safe` transactions and productions (cf. rules on page 13 and rules on page 20), which are part of the same section or its subsections. A global constraint, which is part of the topmost declaration list of a specification is, for instance, valid for all `safe` productions and transactions of the specification.

Global constraints (as well as local constraints) usually are interpreted as static integrity constraints, which are respected by all (`safe`) graph transformations of a given specification. In this case, constraint checking is used for debugging purposes only and may be deactivated without changing the behavior of a correct specification. This interpretation of constraints is no longer valid if repair actions are defined, which are called whenever their constraints are violated. A repair action is a graph transformation, which is always successful and eliminates the detected inconsistency. A violated constraint without repair action as well as a violated constraint with failing repair action terminates execution immediately.

(44) **ConstraintBody** ::=
GraphConstraint | StatExpr

A global constraint may be defined using a graphical notation or a textual notation. The graphical notation has about the same form as the graphical notation of productions, the textual notation is the same as the one used for the definition of complex graph queries. Textually defined global constraints with repair actions usually have about the following form:

```
constraint
  for all P1 := instance of PERSON do
    for all P2 := P1.-child-> do
      ensure
            (P1.Age > P2.Age)
      else
            SomeOptionalRepairAction(P1, P2)
      end
    end
  end;
```

A global constraint without a repair action may be derived from the example above by replacing the `ensure` statement with the repair action by the simple condition `(P1.Age > P2.Age)`.

(45) **GraphConstraint** ::=
[OptConstrLeftSide]
ConstrRightSide
[OptRepairAction]

A graphical constraint definition consists of one or two subgraph patterns, where the first one (the so-called left-hand side) may be omitted, and where the second one (the so-called right-hand side) contains the first one. It has an optional repair action. This repair action is called, whenever a match for the first subgraph pattern is found, which cannot be extended to a match of the second graph pattern. A graphical constraint with an empty left-hand side is a special case of the explanation above. Its empty left-hand side matches all graphs. As a consequence, its right-hand side has to match all graphs of the regarded class of graphs, too.

(46) **OptConstrLeftSide** ::=

```
for   ConstrLeftSideList
    [ OptAttConditionList ]
```

The left-hand side of a graphical constraint is a graph pattern definition accompanied by a list of attribute conditions. It defines a set of subgraphs of the inspected graph, which have to be checked for consistency. The left-hand side

```
for
  begin (* defined using the textual notation *)
    obl node ‘1: MAN; ‘1 -> ‘2: married;
    obl node ‘2: WOMAN;
  end
```

matches, for instance, all married pairs of nodes in the regarded graph and forwards them to the constraint’s right-hand side.

(47) **ConstrRightSide** ::=

```
ensure  ConstrRightSideList
      [ OptAttConditionList ]
```

The right-hand side of a graphical constraint is a graph pattern definition accompanied by a list of attribute conditions. It extends the graph pattern definition of the left-hand side by repeating (at least) the node declarations of the left-hand side. It adds a number of constraints concerning the context and the attribute values of the already determined left-hand side nodes. The right-hand side

```
ensure
  begin (* defined using the textual notation *)
    old obl node ‘1; ‘3 -> ‘1: child;
    old obl node ‘2; ‘3 -> ‘2: child;
    not node ‘3: PERSON;
  end
  condition abs(‘1.Age - ‘2.Age) < 80;
```

extends the above introduced left-hand side (please note that it is not necessary to repeat edges, paths, and restrictions of the left-hand side). It requires that two married nodes do not have a common parent node and that the difference of their `Age` attributes is less than `80`.

(48) **ConstrLeftSideList** ::=

```
begin
  { LeftSideClause }
    LeftSideClause
end
```

The graphical part of a global constraint’s left-hand side. It offers the same elements for the definition of graph patterns as the left-hand sides of productions (cf. rules on page 22), with the exception of optional and set node declarations (which are prohibited by the language’s static semantics).

(49) **ConstrRightSideList** ::=

```
begin
  { ConstrRightSideClause }
    ConstrRightSideClause
end
```

The graphical part of a global constraint’s right-hand side. It offers the same elements for the definition of graph patterns as the left-hand sides of productions (cf. rules on page 24), with the exception of optional and set node declarations (which are prohibited by the language’s static semantics). Furthermore, we need one additional element, which is explained below.

(50) **ConstrRightSideClause** ::=

```
ConstrOldNodeDecl | LeftSideClause
```

The graphical part of a global constraint’s right-hand side defines a new graph pattern, which contains implicitly or explicitly all nodes of the left-hand side pattern. Explicit references to left-hand side nodes are defined using so-called old node declarations.

(51) **ConstrOldNodeDecl** ::=

```
old obl node ApplOldNodeId “;”
```

An old node declaration on a global constraint’s right-hand side refers to a node declaration on its left-hand side. Its identifer is an applied occurrence of the identifier of the left-hand side declaration, and it is a kind of redeclaration for the scope of the right-hand side.

3.9 Declarations of Queries and Transactions

(52) **QueryDecl** ::=
```
  query DeclQueryId [ OptFormParList ]
                    [ Qualifier ] "="
    StatExpr
  end ";"
```
A query is a parametrized subprogram, which calls an arbitrary number of tests but not productions to implement its desired behavior. The optional qualifier indicates whether the execution of the query may fail or always returns successfully and whether it returns a deterministically or nondeterministically computed result. This information is used for consistency checking purposes and for reducing the bookkeeping overhead for backtracking purposes. The default qualifier `[0:n]` defines a possibly failing query with a nondeterministic behavior.

(53) **TransactionDecl** ::=
```
  [ OptSafe ] transaction DeclTransactionId
                                [ OptFormParList ]
                                [ Qualifier ] "="
    [ OptPreCondDecl ]
      StatExpr
    [ OptPostCondDecl ]
  end ";"
```
A transaction is a parametrized subprogram, which calls an arbitrary number of tests and productions to implement the desired graph transformation. The optional qualifier indicates whether the execution of the transaction may fail or always returns successfully and whether it returns a deterministically or nondeterministically computed result. This information is used for consistency checking purposes and for reducing the bookkeeping overhead for backtracking purposes. The default qualifier `[0:n]` defines a possibly failing transaction with a nondeterministic behavior. The optional keyword `safe` marks those productions, which take a consistent graph as input and produce a consistent graph as output with respect to all relevant integrity constraints. All productions without this prefix may produce intermediate graph states, which violate some integrity constraints.

(54) **OptSafe** ::= `safe`

The optional key word `safe` of productions and transactions marks those graph transformations which are expected to return a consistent graph (with respect to all defined integrity constraints) as output. The last action of a safe graph transformation checks all local integrity constraints of node classes (cf. rules on page 10) and all related global constraint declarations (cf. rules on page 11).

(55) **StatExpr** ::=
```
    AndStatList
  | ConcStatList
  | OrStatList
  | StatTerm
```
The body of a query or transaction is either a simple substatement (term) or a list of substatements.

(56) **StatTerm** ::=
```
    EnsureStat
  | ChooseStatList | LoopStatList
  | ForAllStat | UseStat | BracketStat
  | AssignStat | CallStat | CutStat
  | DefStat | NotStat
  | FailStat | HaltStat | SkipStat
  | Factor
```
A simple substatement (term) has either one of the forms defined on the following pages or it is a `boolean` expression (factor). A `boolean` expression, which is evaluated to `true`, is simply skipped, a `boolean` expression, which is evaluated to `false`, triggers backtracking. The subprogram
```
  use i: integer := elem(1 or 2) do
    Trafo(i)
  & (i = 2)
  end
```
assigns either the value `1` or `2` to the variable `i`. The execution with `i = 1` proceeds with some graph transformation `Trafo` and fails in the condition `(i = 2)`. Backtracking cancels all graph modifications of `Trafo` and restarts the evaluation of the subprogram with `i = 2`.

3.10 Pre- and Post-Conditions and Ensure Statements

(57) **OptPreCondDecl** ::=

```
requires
  StatExpr
end;
```

The precondition of a transaction is a query, which should never fail applied to the input graph of the surrounding transaction. It is a `boolean` expression in the simplest case such as

```
requires (Par1.Age > Par2.Age) end.
```

A more complex example of a precondition checks whether it is possible to reduce the input graph with some transaction `ReduceGraphTrafo` into another graph, which fulfills a certain property (like being the empty graph):

```
requires
  def( ReduceGraphTrafo & CheckProperty )
end
```

Please note that such a precondition transforms the input graph for runtime checking purposes, but resets the graph to its previous state before the surrounding transaction's body is entered (cf. rules on page 19). The execution of a complex precondition may take a long time. It is, therefore, possible to deactivate checking of preconditions (together with postconditions and constraints).

A failing precondition stops the execution process immediately.

(58) **OptPostCondDecl** ::=

```
ensures
 StatExpr
end;
```

The postcondition of a transaction is a query, which should never fail applied to the output graph of the surrounding transaction. It is allowed to access the in- and out-parameters of its transaction, but does not distinguish between a before- and a after-state of referenced nodes. The postcondition of the following transaction

```
transaction IncAge(P PERSON; i: integer; out NewP: PERSON) =
  requires (i > 0) end
  begin
    P.Age := (P.Age + i)
  & NewP := P
  end
  ensures NewP.Age = (P.Age + i) end
end;
```

would always fail. The in-parameter `P` references at the end of the transaction the same node as the out-parameter `NewP`. As a consequence, `NewP.Age` and `P.Age` yield the same result. It is the subject of future research to allow the comparison of input graph states with output graph states in postconditions of transactions.

A failing postcondition stops the execution process immediately.

(59) **EnsureStat** ::=

```
ensure
  StatExpr
else
  StatExpr
end
```

An `ensure` statement may be used within textually defined integrity constraints only (cf. rules on page 10 and rules on page 11) to define a repair action. The explanation of global constraints already presents one example of such a repair action. Please note that repair actions have to be deterministic and total (never failing) graph transformations, which remove the triggering inconsistency. Repair actions may be used to write specifications, which have the same flavor as rule-oriented active database systems.

The usage of `ensure` statements in the body of a query or transaction and the pre- or postconditions of a transaction is strictly prohibited.

3.11 Statement Lists

(60) **AndStatList** ::=
`{ StatTerm and } StatTerm`

The and statement requires that all its substatements are executed, but makes no requirements concerning the order of execution. It is, therefore equivalent to a combination of the nondeterministic or statement and the deterministic concatenation & :

$\texttt{Stat}_1$ and $\texttt{Stat}_2$ = ($\texttt{Stat}_1$ & $\texttt{Stat}_2$) or ($\texttt{Stat}_2$ & $\texttt{Stat}_1$).

The semideterministic execution mode executes first the listed substatements in their given order (from left to right). If this execution order fails or causes failure of a subsequent graph transformation step then another permutation is selected and executed. The random execution mode starts with the execution of a randomly selected permutation of the substatements. The default execution mode is semideterministic, which simplifies debugging of specifications.

Finally note that the and statement may be read as a boolean operator if all its substatements are not graph transformations but graph queries.

(61) **ConcStatList** ::=
`{ StatTerm & } StatTerm`

The concatenation statement executes its substatements in the given order. It succeeds if all its substatements succeed, it fails without any graph or variable modifications (and triggers backtracking) if any of its substatements fails. This is even true if the last substatement fails and the preceding substatements were arbitrarily complex graph transformations.

The & operator behaves like a boolean and with short-circuit (left to right) evaluation if all its substatements are not graph transformations but graph queries.

(62) **OrStatList** ::=
`{ StatTerm or } StatTerm`

The or statement selects and executes one of its substatements. If the selected substatement fails or causes failure of a subsequent graph transformation then backtracking reenters the or statement, cancels the effects of the executed substatement, and proceeds with another still remaining substatement. The or statements fails as a whole if all its substatements fail. The semideterministic execution mode selects substatements in the given order (from left to right), the random mode starts with the execution of a randomly selected substatement. The default execution mode is semideterministic, which simplifies debugging of specifications.

The subprogram

```
use i, j := 2 do
   begin
      j := j * 2
    & (i = j)
   end
or
   begin
     j := j * 3
   end
end
```

computes, therefore, always the value 6 for j and the value 2 for i, although its execution usually starts with the first substatement and the assignment of the value j * 2 = 4 to j (backtracking restores the old value 2 of j before entering the second substatement).

Finally note that the or statement may be read as a boolean operator if all its substatements are not graph transformations but graph queries.

3.12 Conditional and Iterating Statements

(63) **ChooseStatList** ::=
choose { CondStat else } CondStat end

The choose statement generalizes the if-then-else statement of other programming languages and Dijkstra's guarded commands. It inspects its substatements from left to right and executes the first branch, which is unguarded and does not fail or which has a valid guard. The main difference between a simple choose statement (with two unguarded branches) such as

```
   choose i := elem(3 or 4) else i := 5 end
& (i = 5)
```

and a similar or statement

```
   begin i := elem(3 or 4) or i := 5 end
& (i = 5)
```

is as follows: The choose statement executes its first branch and assigns the value 3 or 4 to i. The following boolean condition fails and triggers backtracking. Backtracking reenters the first branch and assigns the remaining value 4 or 3 to i. The boolean condition fails again and causes failure of the whole subprogram, i.e. backtracking does not enter a branch of the choose statement with a preceding (initially) successfully executed branch. The or statement, on the contrary, allows backtracking to enter its second branch after all possible execution paths of its first branch failed.

The execution of a choose statement fails if all its guarded branches have invalid guards and if all its unguarded branches fail or if the execution of a branch with a valid guard fails. The subprogram

```
choose
  when true then fail
else
  skip
end
```

is, therefore, an example of an always failing choose statement.

(64) **LoopStatList** ::=
loop { CondStat else } CondStat end

The loop statement is an iterated choose statement. It executes its first branch (form left to right), which is unguarded and does not fail or which has a valid guard. It terminates successfully if all its unguarded branches fail and all its guarded branches have an invalid guard. It fails as a whole (without any graph modifications) if the execution of one of its guarded branches fails although the guard of this branch is valid. The loop

```
loop
  when i < 0 then fail
  when i < n then DoSomething(i) & i := (i+1)
end
```

is an example of a loop statement, which fails and triggers backtracking for an initial i value smaller than 0 and which executes DoSomething until i = n. The loop terminates successfully if DoSomething is always executable, it fails otherwise.

(65) **CondStat** ::=
GuardStat | StatExpr

The branches of a choose statement or a loop statement is a guarded or an unguarded substatement. An unguarded substatement such as

```
... else DoSomething else ...
```

should be partial (except for the last branch of a choose statement), i.e. static analysis should not be able to guarantee its successful execution (a never failing branch blocks the execution of all remaining branches and prohibits termination of its loop). Furthermore, it may be regarded as an abbreviation for (cf. rules on page 19)

```
... else when def(DoSomething) then DoSomething else ...
```

(66) **GuardStat** ::=
when StatExpr then StatExpr

A guarded branch of a choose or loop statement has a a graph query (or boolean expression) as its first component (guard). Static analysis guarantees that the execution of such a query has no side-effects. The guarded branch is skipped if the query fails, its second component is executed otherwise.

3.13 Bounded Iteration and Variable Declarations

(67) **ForAllStat** ::=

```
for all LocalsList do
  StatExpr
end
```

The `for all` statement is a loop which assigns one possible value after the other to its variables and executes its body for each possible combination of variable bindings. The `for all` statement

```
for all i := elem(1 or 2); j := elem(3 or 4) do ... end
```

executes, therefore, its body four times for `i = 1` and `j = 3`, `i = 1` and `j = 4` etc.

The `for all` statement should only be used (instead of the `loop` statement) if the order in which values are assigned to variables does not matter. The subprogram

```
    for all i := elem(Set) do
      k := k + i
    end
  & fail
```

fails without reinspecting the body of the loop, whereas

```
    loop
      i := elem(Set) (* terminates loop for empty Set *)
    & Set := (Set but not i)
    & k := k + i
    end
  & fail
```

backtracks over all possible permutations of the order in which elements are extracted from the given input set `Set`.

(68) **UseStat** ::=

```
use LocalsList do
  StatExpr
end
```

A `use` statement allows to introduce local variables in the body of a transaction or a query, wherever these variables are needed. It is a good programming style to make the scope of a local variable as small as possible. The semantics of a `use` statement is about the same as the semantics of a `use` expression (cf. rules on page 41) with the following two exceptions: (1) a local variable inside an expression may not change its initial value, a local variable of a subprogram (statement list) may change its value an arbitrary number of times. (2) The assignment of a set of values to a single element containing variable is treated differently. A `use` expression evaluates its body for the assignment of any possible value in the set to its variable, a `use` statement selects one possible assignment and goes ahead. Backtracking may be used to select another variable and to recompute all following statements. The transaction

```
transaction T(out P: integer) =
  use i: integer := elem(1 or 2 or 3); j: integer do
    j := elem(1 or 2 or 3)
  & (i = j)
  & P := i * j
  end
end
```

is a good example of a backtracking graph transformation. It retunrs the value 1or 6 or 9 for its out-parameter `P`. Its execution proceeds as follows:

(1) The initial value `1` or `2` or `3` is assigned to the variable i.
(2) The value `1` or `2` or `3` is assigned to the variable j .
(3) The condition `(i = j)` triggers backtracking and variable reassignments until it is fulfilled.
(4) The out-parameter `P` receives the value `1` or `4` or `9`.

The following subprogram might be used to reject the nondeterministically computed result of transaction `T` via backtracking until it returns the possible value `9` or `4` (and not `1`):

```
use v: integer do
  T(out v)
& (v > 2)
end
```

3.14 Graph Modifying Statements

(69) **BracketStat** ::=
`begin StatExpr end`

The keywords `begin` and `end` are the delimiters of a block of statements. They allow the definition of a complex `StatExpr`, where a simple `StatTerm` is required.

(70) **AssignStat** ::=
`Destination ":=" Factor`

An assign statement is usually used to define a (new) value of a local variable or an out-parameter. But it may be (mis-)used to change the attribute value of a given node. Otherwise, one would be forced to write a production, which matches the regarded node with its left-hand side, preserves this node, and uses its attribute `transfer` part to assign the new value to the node's attribute (cf. rules on page 26).

(71) **Destination** ::=
`ApplVarOutParId | SelectDestination`

The destination on the left-hand side of an assign statement either is a variable or an out-parameter (but not an in-parameter) or a node attribute.

(72) **SelectDestination** ::=
`ApplVarInParId "." ApplIntrinsicAttId`

A node attribute is determined by a variable or in-parameter, which contains a reference to a node, followed by the name of an attribute. This attribute has to be `intrinsic` and it should not be a pointer to a node, i.e. an abbreviated edge type declaration (cf. rules on page 7) . Node attributes, which do not contain simple values but point to other nodes, have to be manipulated like edges (on the left- and right-hand sides of productions).

(73) **CallStat** ::=
`ApplActionId [ OptActParList ]`

This is the call of a test or a query or a production or a transaction or a `boolean` function or an applied occurrence of a local variable or in-parameter of type `boolean`. A `boolean` expression, which is evaluated to `true`, is simply skipped, a `boolean` expression, which is evaluated to `false`, triggers backtracking. A called test or query may succeed or fail, and it may modify the state of a number of local variables or out-parameters. A called production or transaction may succeed or fail, and it may modify the state of the processed graph as well as the values of a number of local variables or out-parameters.

(74) **CutStat** ::=
`StatTerm ":" Qualifier`

The cut statement allows to cast a potentially failing graph transformation into a total operation. Furthermore, it may be used to prune the search tree of a nondeterministic graph transformation. The statements

`PartialTrafo : [1:n]` or `PartialTrafo : [1:1]`

require that the regarded graph transformation does not fail; they stop the execution process instead of starting backtracking if the graph transformation fails nevertheless. The statements

`NondeterministicTrafo : [0:1]` or `NondeterminsticTrafo : [1:1]`

are similar to the cut statement of Prolog. As soon as their execution is completed they are never re-entered during backtracking. The internally maintained search tree is pruned such that backtracking skips the regard graph transformation.

3.15 Graph Query Statements

(75) **DefStat** ::=
`def "(" StatExpr ")"`

The `def` statement tests whether its argument, an arbitrarily complex graph transformation, would be executable. It executes its substatement and fails if the execution of the substatement fails. It succeeds without modifying its input graph if the execution of its substatement succeeds (even if the substatement transforms the input graph).

The argument of a `def` statement often is a single production, but it could also be a very complex graph transformation followed by complex graph query as in:

```
def( ReduceGraphTrafo & CheckProperty ) .
```

Such a `def` statement is usually used as the guard of one branch of a `choose` statement (cf. rules on page 16) or a `loop` statement. It allows the execution of a certain graph transformation if the input graph could be reduced to another graph with certain properties, the execution of another graph transformation otherwise.

(76) **NotStat** ::=
`not StatTerm`

The `not` statement has a side-effect-free graph query as its argument. It inverses the effect of the graph query, i.e. it fails if the graph query succeeds, it succeeds if the graph query fails. Subprograms of the form

```
not Transaction(...)     or    not Query(out v)
```

are not allowed, since they would either modify their input graph or the values of some variables. Please note that it is possible to write

```
not def(AnyGraphTransformation)
```

i.e. to go ahead if a certain graph transformation fails and to backtrack if the regarded graph transformation would succeed.

(77) **FailStat** ::= `fail`

The `fail` statement is the always failing graph transformation.

(78) **HaltStat** ::= `halt`

The `halt` statement stops the execution process immediately. It is often used as the last branch of a `choose` statement. The subprogram

```
choose DoSomething else halt end
```

is equivalent to

```
choose DoSomething end : [1:n].
```

(79) **SkipStat** ::= `skip`

The `skip` statement is the always successful graph transformation. It does not modify its input graph and is equivalent to

```
not fail.
```

It is often used as the last branch of a `choose` statement such that a may be failing graph transformation is executed if possible and skipped otherwise:

```
choose DoSomething else skip end .
```

3.16 Declarations of Tests and Productions

(80) **TestDecl** ::=
```
  test DeclTestId
       [ OptFormParList ]
       [ OptPTQualifier ] "="
    TestBody
  end ";"
```
A test is a short-hand for a production with the same left- and right-hand side. It either finds a match for its left-hand side or fails. It may or may not return the identifiers or attribute values of matched nodes via its out-parameters. Its in-parameters are used to fix certain nodes of the matched subgraph or to require additional attribute conditions. The additional qualifier determines the number of expected and processed matches of the test's left-hand side in a graph (with `[0:n]` as default).

(81) **TestBody** ::=
```
  [ OptLeftSideList ]
  [ OptFoldList ]
  [ OptAttConditionList ]
  [ OptReturnList ]
```
A test searches for a subgraph, which is in the usual case isomorphic to its left-hand side. The given folding list allows some left-hand side nodes to share their matches. The attribute condition list introduces requirements for matched nodes, and the test's return list assigns appropriate values to all out-parameters.

(82) **ProductionDecl** ::=
```
  [ OptSafe ] production DeclProductionId
                         [ OptFormParList ]
                         [ OptPTQualifier ] "="
      ProductionBody
  end ";"
```
Productions are the basic programming constructs. They match and modify subgraphs of a graph in one indivisible rewrite step or fail as a whole without causing any graph modifications. A production may return the identifiers or attribute values of matched nodes or new nodes via its out-parameters. Its in-parameters are used to fix certain nodes of the matched subgraph or to require additional attribute conditions. The additional qualifier determines the number of expected and processed matches of the production's left-hand side in a graph (with `[0:n]` as default). The optional keyword `safe` marks those productions, which take a consistent graph as input and produce a consistent graph as output with respect to all relevant integrity constraints (cf. rules on page 13). All productions without this prefix may produce intermediate graph states, which violate some integrity constraints.

(83) **ProductionBody** ::=
```
    GraphPart
  [ OptFoldList ]
  [ OptAttConditionList ]
  [ OptEmbeddingList ]
  [ OptAttTransferList ]
  [ OptReturnList ]
```
A production searches for a subgraph, which is in the usual case isomorphic to its left-hand side. The given folding list allows some left-hand side nodes to share their matches. The attribute condition list introduces requirements for matched nodes. The embedding list may be used to manipulate edge bundles of undetermined size between the rewritten subgraph and its direct context nodes. The attribute transfer list is needed to assign new values to preserved or created nodes. The production's return list finally assigns appropriate values to all out-parameters.

(84) **GraphPart** ::=
```
    [ OptLeftSideList ]
  ::=
    [ OptRightSideList ]
```
A production's left- and right-hand side. Any node, which is part of the left-hand side but not bound to a node of the right-hand side, deletes its match in the given graph. The same is true for any edge, which is part of the left-hand side but not repeated in the right-hand side. Any node, which is a new node in the right-hand side, creates a copy of itself in the given graph. The same is true for any edge, which is part of the right-hand side but not of the left-hand side.

3.17 Test and Production Qualifiers

(85) **OptPTQualifier** ::= Star | Plus | Qualifier

The qualifiers of tests and productions have two functions: (1) * and + enforce parallel processing of all left-hand side matches, whereas (2) all remaining qualifiers are an indicator for the number of expected matches.

[0:1] indicates that there is at most one left-hand side match, we are interested in.
[1:1] indicates that there is always one left-hand side match, we are interested in.
[1:n] indicates that there is at least one left-hand side match, we are interested in.
[0:n] is the default value; it has no semantic changing effects

The term "we are interested in" above means that a left-hand side may have more than one match in a given graph, but only one of these possible matches will be processed, even if backtracking occurs. The qualifiers [0:1] and [1:1] have, therefore, about the same semantics as the cut in Prolog. They prune the regarded search space after the first successful match. The qualifiers [1:1] and [1:n] have the (additional) effect to stop the program's execution (instead of starting backtracking) if the regarded test or production fails. The test

```
test AnyPerson( out P : Person ) [1:1] =
  begin ‘1 : PERSON; end (* defined using the textual notation *)
  return P := ‘1;
end;
```

stops execution (with run-time error) if its input graph does not contain a single Person node. It returns a randomly selected Person node otherwise. Backtracking is not allowed to reenter the test's body to determine another node match if needed, but returns to a previously made nondeterministic decision of another test or production.

(86) **Star** ::= "*"

The qualifier * causes the application of a test or production to all its matches in parallel. This includes the empty set of matches as a special case, i.e. a * marked test or production never fails. Out-parameters of * marked tests or productions must be sets. They collect the set of all assignments of all processed matches. The test

```
test AnyPerson( out P : Person [0:n] ) * =
  begin ‘1 : Person; end (* defined using the textual notation *)
  return P := ‘1;
end;
```

returns e.g. the set of all Person nodes in a given graph.

The result of the parallel application of a production to all its matches is undetermined if these matches overlap and if the inspected nodes, edges, and attribute values of overlapping matches are not preserved (or modified in contradicting ways). This condition is neither checked at compile-time nor at run-time due to the difficulty to determine the effects of a production onto derived attributes and relationships. The parallel execution of a production is not equivalent to the repeated sequential execution of a production. The iterated sequential application of

```
production CreateChild =
  begin (* defined using the textual notation *)
    ‘1 : Person;
  end
::=
  begin
    1’ = ‘1; 1’ -> 2’: child; 2’ : Person;
  end
  transfer 1’.NoOfChildren := ‘1.NoOfChildren + 1;
end;
```

is a nonterminating graph rewriting process, which creates an unbounded number of Person nodes with an unbounded number of child nodes. The parallel application of the same production adds to each Person node in the graph exactly one child Person node.

(87) **Plus** ::= "+"

The qualifier + causes the parallel application of a test or production with at least one match. The marked test or production fails if it finds no match in the graph.

3.18 Left-hand Sides of Productions and Graph Patterns of Tests, etc.

(88) **OptLeftSideList** ::=
```
begin
  { LeftSideClause }
    LeftSideClause
end
```
A left-hand side list defines the graph pattern of a production, test, path, or restriction we are looking for. It usually is a nonempty list of pattern defining clauses. The order of list elements is of no importance.

(89) **LeftSideClause** ::=
```
   OblNodeDecl | OptNodeDecl | OptSetDecl | OblSetDecl| NotNodeDecl
| EdgeDecl | NotEdgeDecl | PathCond | NotPathCond | RestrictCond
```
All constructs which may be used to define graph patterns, i.e. left-hand sides of productions etc.

(90) **OblNodeDecl** ::=
```
obl node DeclNodeId NodeDescription ";"
```
An obligate node declaration has a solid rectangle as its graphical representation. It matches a single node of the regarded graph, which fulfills all required conditions.The matched node is randomly selected from the set of all possible candidates. Backtracking may be used later on to withdraw the made selection and to match another node from the reduced candidate set. An empty candidate set causes failure of the overall pattern matching process.

(91) **OptNodeDecl** ::=
```
opt node DeclNodeId NodeDescription ";"
```
An optional node declaration has a dashed rectangle as its graphical representation. It matches a single node of the regarded graph if possible, the undefined node `nil` otherwise. The matched node is randomly selected from the set of all possible candidates. An empty candidate set does not cause the overall pattern matching process to fail. Backtracking may be used later on to withdraw the made selection and to match another node from the reduced candidate set.

(92) **OblSetDecl** ::=
```
obl set DeclNodeId NodeDescription ";"
```
An obligate set node declaration has a solid double rectangle (rectangle with shadow) as its graphical representation. It matches the maximum set of possible candidates, which fulfill all required conditions. An empty set of candidates triggers failure of the overall pattern matching process.

(93) **OptSetDecl** ::=
```
opt set DeclNodeId NodeDescription ";"
```
An optional set node declaration has a dashed double rectangle (rectangle with shadow) as its graphical representation. It matches the maximum set of possible candidates, which fulfill all required conditions. This set may be empty without causing failure of the overall pattern matching process.

(94) **NotNodeDecl** ::=
```
not node DeclNodeId NodeDescription ";"
```
A negative node declaration has a crossed-out solid rectangle as its representation. It is first treated in the same manner as an obligate node by the pattern matching process. But its effects onto the pattern matching process are then complemented. A negative node which matches a node of the regarded graph leads to a failure of the pattern matching process, a negative node without a match is simply ignored afterwards.

(95) **NodeDescription** ::=
```
[ ":" Factor ] [ "=" Expression ]
```
This construct determines the required type(s) and/or identity of matched nodes. The optional factor following ":" computes the set of types, whose instances are possible candidates for the matching process. The optional expression following "=" determines a set of nodes or a single node as the regarded candidate set. Important special cases are:

- `'n : Type` : node instances of type `Type` are regarded.
- `'n : Class` : node instances of any type which belongs to class `Class` are regarded.
- `'n : 'm.type` : `'n` matches a node of the same type as `'m`.
- `'n = InPar` : the parameter `InPar` determines the regarded match of `'n`.
- `'n = elem(InSet)` : the parameter `InSet` determines the regarded candidate set.

(96) **EdgeDecl** ::=
`ApplNodeId “->” ApplNodeId “:” ApplEdgeTypeId “;”`

An edge declaration has a solid arrow as its graphical representation, which starts at a given source node and ends at a given target node. It requires the existence of an edge of the given type between the determined source and target nodes.

(97) **NotEdgeDecl** ::=
`ApplNodeId “+>” ApplNodeId “:” ApplEdgeTypeId “;”`

A negative edge declaration has a crossed-out solid arrow as its graphical representation, which starts at a given source node and ends at a given target node. It requires the nonexistence of an edge of the given type between the determined source and target nodes.

(98) **PathCond** ::=
`ApplNodeId “=>” ApplNodeId “:” OpExpr “;”`

A path condition has a double arrow as its graphical representation, wich starts at a given source node and ends at a given target node. It requires the existence of a certain path of edges between the determined source and target node or, more general, the existence of a given derived relationship between them. The condition

```
‘1 => ‘2 : (-contains-> or <-contains-)+
```

requires, for instance, the existence of a path of `contains` edges from the match of `‘1` to the match of `‘2`. The direction of regarded `contains` edges does not matter.

Please do not use (very) complex path expressions (`OpExpr`) directly as path conditions. It is better to introduce these path expressions separately in the form of path declarations (cf. rules on page 28) and to replace the complex text expression of the path condition by the path declaration’s identifier.

(99) **NotPathCond** ::=
`ApplNodeId “#>” ApplNodeId “:” OpExpr “;”`

A negative path condition has a crossed-out double arrow as its graphical representation, wich starts at a given source node and ends at a given target node. It requires the nonexistence of a certain path of edges between the determined source and target node or, more general, the nonexistence of a given derived relationship between them.

(100) **RestrictCond** ::=
`“=>” ApplNodeId “:” OpExpr “;”`

A restrict condition is a double arrow, which points to a target node but has no source node. It imposes an additional restriction onto the candidate set (matches) of its target node. It allows one to move simple attribute conditions from the textual condition part of productions, tests, etc. into the graphical representation of the defined pattern. Furthemore, restrict conditions may be used (in contrast to textual attribute conditions) to require certain properties of optional nodes, set nodes, and negative nodes. Typical examples of this kind are

```
=> ‘n : not with (<-contains- or -contains->) ;
```

for node `‘n` must not have an incoming or outgoing `contains` edge or

```
=> ‘n : valid ( self.Att = ‘m.Att ) ;
```

for node `‘n` (which is `self` in the restriction’s select expression) has the same `Att` attribute value as node `‘m`.

Please do not use (very) complex restrict expressions (`OpExpr`) directly in graph pattern definitions. It is better to introduce these expressions separately in the form of restriction declarations (cf. rules on page 29) and to replace the complex text expression of the restrict condition by the restriction declaration’s identifier.

(101) **DeclNodeId** ::= ...

This is the identifier of a node on a production’s left-hand side (or in the graph pattern of a test, path declaration etc.). It has “`‘`” as the leading character. Usually, node identifiers have the form `‘n` with `n` being a cardinal number.

3.19 Right-hand sides of Productions

(102) **OptRightSideList** ::=
```
begin
  { RightSideClause }
    RightSideClause
end
```
A production's right-hand side defines the graph pattern which replaces the match of its left-hand side. It usually is a nonempty list, where the order of list elements is of no importance.

(103) **RightSideClause** ::=
```
  OldOblNodeDecl | OldOptNodeDecl | OldOblSetDecl | OldOptSetDecl
| NewNodeDecl | NewEdgeDecl
```
A production's right-hand side consists of old node declarations and new node as well as edge declarations. Old node declarations are shared with the production's left-hand side, whereas new node and edge declarations belong to its right-hand side only. Any node declaration on a production's left-hand side is bound to at most one old node declaration on its right-hand side of the same kind. Matches of left-hand side node declarations with corresponding right-hand side declarations are preserved, all remaining matches of left-hand side nodes are deleted. Any new node and edge declaration leads to the creation of a new node or edge of the given type in the regarded graph.

(104) **OldOblNodeDecl** ::= `obl node` DeclNewNodeId "=" ApplOldNodeId ";"

An old obligate node declaration is bound to an obligate node declaration of the production's left-hand side. It requires the existence of a node with certain properties and preserves this node.

(105) **OldOptNodeDecl** ::= `opt node` DeclNewNodeId "=" ApplOldNodeId ";"

An old optional node declaration is bound to an optional node declaration of the production's left-hand side. It requires that the matched node has to be preserved.

(106) **OldOblSetDecl** ::=
`obl set` DeclNewNodeId "=" ApplOldNodeId ";"

An old obligate set declaration is bound to an obligate set declaration of the production's left-hand side. It requires the existence of a nonempty set of nodes with certain properties and preserves all nodes in this set.

(107) **OldOblNodeDecl** ::=
`opt set` DeclNewNodeId "=" ApplOldNodeId ";"

An old optional set declaration is bound to an optional set declaration of the production's left-hand side. It requires that the matched possibly empty set of nodes is preserved.

(108) **NewNodeDecl** ::= DeclNewNodeId ":" Factor ";"

A new node declaration requires the creation of a node of a given type `T`. This type `T` is determined by evaluating the expression following ":". The result of this expression has to be a single well-defined node type.

(109) **NewEdgeDecl** ::= ApplNewNodeId "->" ApplNewNodeId ":"  lEdgeTypeId ";"

A new edge declaration requires the creation of a new edge between the given two new or preserved nodes. Parallel edges of the same type are identified. Please note that matches of left-hand side edge declarations are always deleted from a theoretical point of view. Corresponding new edge declarations on the right-hand side are necessary to recreate (preserve) them.

(110) **DeclNewNodeId** ::= ...

This is the identifier of a node on a production's right-hand side. It has "'" as the trailing character. Usually, new node identifiers have the form `n'` with `n` being a cardinal number.

(111) **ApplOldNodeId** ::= ...

This is the applied occurrence of a node identifier declared in a production's left-hand side. It has a "'" as the leading character. Usually, old node identifiers have the form `'n` with `n` being a cardinal number. It is common sense to use the same number `n` in related node declarations of left- and right-hand sides. This leads to old node (set) declarations like

```
... n' = 'n ;
```

3.20 Node Foldings and Attribute Conditions

(112) **OptFoldList** ::=
folding { NodeIdList } NodeIdList

Folding clauses may be used to deactivate the default requirement that two different nodes of a test's or production's left-hand side match two different graph nodes. A folding clause of the form

folding {‘1,‘2}; {‘3,‘4};

allows for instance that ‘1 and ‘2 or ‘3 and ‘4 are mapped onto the same node. It still disallows that ‘1 and ‘3 or ‘2 and ‘3 etc. are mapped onto the same node.

Please note that the current language version disallows the occurrence of one left-hand side node in two folding sets, i.e. folding instructions like

folding {‘1,‘2}; {‘1,‘3};

are not (yet) supported.

(113) **NodeIdList** ::=
“{” { ApplNodeId “,” } ApplNodeId “}” “;”

A folding set defines a list of left-hand side nodes which are allowed to match the same graph node. The order of list elements has no significance. All left-hand side nodes of one folding set have to possess a common superclass; otherwise, sharing of nodes would be impossible. Furthermore, all referenced left-hand side nodes have to be preserved nodes if the folding clause is part of a production and not of a test. Otherwise, it would be possible to "fold" a perserved node ‘1 (which has a counterpart 1’=‘1 on the production's right-hand side) with a to be deleted node ‘2 (without such a counterpart on the right-hand side), thereby creating a conflict between node preservation and node deletion.

(114) **OptAttConditionList** ::=
condition { AttCondition } AttCondition

A list of attribute conditions restricts the possible set of matches of referenced nodes. The order of list elements has no significance. Furthermore, it makes no difference for the semantics of a specification whether we write

condition $AttCond_1$ and $AttCond_2$;

or

condition $AttCond_1$; $AttCond_2$;

But the second variant listed above may lead to a more efficient search plan. The first variant forces the pattern matcher to test both attribute conditions together, whereas the second variant gives the pattern matcher the freedom to insert other pattern matching steps between the two listed attribute conditions. Furthermore, the pattern matcher is able to recognize

condition ‘1.IndexAtt = $expr_1$; ‘2.IndexAtt = $expr_2$;

but not

condition (‘1.IndexAtt = $expr_1$) and (‘2.IndexAtt = $expr_2$);

as two index attribute conditions, which may be used to determine appropriate matches for ‘1 and ‘2 rather efficiently by accessing the internally maintained attribute indexes.

(115) **AttCondition** ::=
Expression “;”

An attribute condition restricts the possible set of matches of its referenced nodes. A referenced node has to be an obligate node in the graph pattern of the surrounding test, production, etc. Attribute conditions for optional nodes or node sets have to be defined as restrictions, which are attached to a single node declaration of this kind (cf. rules on page 23). Please note that the name “attribute condition” is a little bit misleading. The expression syntax (cf. rules on page 36) allows one to define conditions concerning the types of referenced nodes or their context. It is e.g. possible to require that a node ‘1 belongs to a class PERSON or that a father edge starting at node ‘1 leads to another node x and that this node x is the source of a mother edge, which has another node ‘2 as target:

```
‘1.type is instance of PERSON;
use x := ‘1.-father-> :: ‘2.<-mother- = x end;
```

These attribute conditions are good examples for the misuse of expressions. A graphical definition of the needed constraints as part of a production's left-hand side would be much more readable.

3.21 Attribute and Return Assignments

(116) **OptAttTransferList** ::=
`transfer { AttTransfer } AttTransfer`

The attribute tramsfer list allows one to assign (new) attribute values to intrinsic attributes of new nodes or preserved nodes of a production. An attribute's new value may not depend on the new values of other node attributes. This is the reason why the order of attribute transfers is of no importance.

(117) **AttTransfer** ::=
`ApplNewNodeId "." ApplIntrinsicAttId ":=" Expression ";"`

An attribute transfer assigns a new value to an intrinsic attribute of a node, which was created or preserved by the surrounding production. The expression on the right-hand side of the assignment may only contain references to nodes,which are preserved or deleted by the production. The expression is evaluated after a match of the production's left-hand side has been determined and before any graph modifications are carried out. Assuming e.g. that two left-hand sides nodes `'1` and `'2` are preserved as right-hand side nodes `1'` and `2'`, respectively, the following attribute transfers

```
transfer 1'.Att := '2.Att; 2'.Att := '1.Att;
```

simply exchange the attribute values of the two referenced nodes.

Please note that the current language version treats references to set nodes or optional nodes on the left- and the right-hand side of attribute transfers differently. An attribute transfer of the form

```
transfer SetNode'.Att := 'SetNode.Att;
```

computes first the union of all `Att` attribute values of all matches of `'SetNode`. This set of attribute values is then assigned to the `Att` attributes of all `SetNode'` matches. Assume e.g. that `SetNode'='SetNode` matches two nodes `n1` and `n2` and that

```
n1.Att = {1,2}   and   n2.Att = {3,4}
```

before the application of the surrounding production. The execution of the previously defined attribute transfer delivers the new attribute values

```
n1.Att = {1,2,3,4}   and   n2.Att = {1,2,3,4}.
```

(118) **OptReturnList** ::=
`return { ReturnTransfer } ReturnTransfer`

A test or production has to possess exactly one return transfer for each of its out-parameters. These return transfers (out-parameter assignments) are evaluated as the last step of the execution of their surrounding test or production. The expression on the right-hand side of a return transfer may not contain occurrences of out-parameters. As a consequence, the list of return transfers may be evaluated in any order.

(119) **ReturnTransfer** ::=
`ApplOutParId ":=" Expression ";"`

The expression, which determines an out-parameters value, may either depend on the state of the graph before the execution of any graph modifications or on the state of the graph after the execution of any graph modifications. The following assignments are legal examples of return transfers:

```
OutType := '1.type;          OutValue := '1.Att + InPar;
OutValue := 1'.Att + InPar;  OutNodes := 1' or 2';
```

The expressions of the first two assignments above are evaluated before the execution of any graph modifications, the remaining two assignments after the execution of any graph modifications.
The value-defining expression of a return transfer may not reference old and new graph states simultaneously, as e.g. in:

```
OutValue := '1.Att - 1'.Att;
```

Return transfers of tests or productions, which process all their matches in parallel, require a special treatment. They are evaluated for each match and assign the union of all computed values to their out-parameters. The test

```
test T(out ValueSet: integer [0:n]) * =
  begin obl node '1 : ITEM; end (* defined using the textual notation *)
  return ValueSet := '1.Att;
end;
```

assigns, for instance, the set of all `Att` attribute values of all `ITEM` nodes in the graph to its out-parameter `ValueSet`.

3.22 Embedding Transformation Clauses

(120) **OptEmbeddingList** ::=
`embedding { Embedding } Embedding`

Textually defined embedding clauses are the appropriate means for manipulating sets of edges between matched nodes and their direct context nodes of unrestricted size if and only if the usage of set nodes for this purpose leads to left- and right-hand sides of unreasonable size (cf. rules on page 22).

(121) **Embedding** ::= `Copy | Remove | Redirect`

There are three different kinds of embedding clauses (rules) for copying, redirecting, and removing context edge bundles, which have matches of left-hand side nodes as source (target) and context nodes (nodes which are not matched by the production's left-hand side) as target (source).

(122) **Copy** ::=
`copy ModifiedEdgeTypeOpList from ApplNodeId to ApplNewNodeId ";"`

This embedding clause allows to create context edges between a preserved node and another preserved node or a new node. The clause

```
copy -child-> as <-stepfather- from '1 to 2';
```

creates e.g. `stepfather` edges from a set of nodes `S` as sources to the match of node `2'` as target. `S` is the set of all those nodes, which are the targets of `child` edges emanating from the match of node `'1` (before the execution of any graph modifications).

(123) **Remove** ::=
`remove EdgeTypeOpList from ApplNodeId ";"`

This embedding clause allows to delete old context node edges. The clause

```
remove -child->, <-stepfather- from '1;
```

deletes for instance all `child` edges, which have node `'1` as source, and all `stepfather` edges, which have node `'1` as target. Please note that a `remove` embedding clause affects only old context edges, which were already existent before the execution of any graph modifications. The combination of the two embedding clauses

```
copy -child-> from '1 to 1'; remove -child-> from '1;
```

with `1'='1` deletes first all `child` edges emanating from node `'1` and recreates them afterwards.

(124) **Redirect** ::=
`redirect ModifiedEdgeTypeOpList from ApplNodeId to ApplNewNodeId ";"`

The redirect embedding clause is just a short-hand for a `remove` and a `copy` clause with the same arguments. The following instruction

```
redirect -child-> as <-stepfather- from '1 to 2';
```

is e.g. an abbreviation for

```
remove -child-> from '1; copy -child-> as <-stepfather- from '1 to 2';
```

(125) **ModifiedEdgeTypeOpList** ::=
`{ ModifiedEdgeTypeOp "," } ModifiedEdgeTypeOp`

Defines a list of context edges (types and directions), which are affected by the enclosing copy or redirect clause. The list may contain instructions, which change the type or direction of context edges.

(126) **ModifiedEdgeTypeOp** ::= `EdgeTypeOp [ OptModifier ]`

The `EdgeTypeOp` part defines the type and direction of the old context edge, the `OptModifier` part the direction and type of the new context edge. The default for a missing `OptModifier` is to preserve the direction and the type of the regarded context edges.

(127) **OptModifier** ::= `as EdgeTypeOp`

The modifier defines the direction and the type of a to be created context edge. It has the form
`-e->` for an edge of type `e`, which has a given right-hand side node as source
`<-e-` for an edge of type `e`, which has a given right-hand side node as target.

(128) **EdgeTypeOpList** ::=
`{ EdgeTypeOp "," } EdgeTypeOp`

Defines a list of context edges, which are affected by the enclosing `remove` clause. Terms such as
`-e->` match all context edges of type `e`, which have a given left-hand side node as source
`<-e-` match all context edges of type `e`, which have a given right-hand side node as target.

3.23 Path Declarations

(129) **PathDecl** ::=
```
[ OptStaticOrVirtual ] path DeclPathId
                           [ OptFormParList ] ":" RelType "="
   PathBody
end ";"
```

A path declaration is the functional abstraction of a path expression. It defines a derived binary relationship between nodes. Its declaration has the same elements as an edge type declaration plus a number of additional elements. A path declaration may, for instance, be used to introduce a (role) name for the reverse traversal of the edges of a certain type:

```
path Husband: Woman [0:1] => Man [0:1] = <-Wife- end;
```

A path declaration may be parametrized with a number of in-parameters. These in-parameters are often node type parameters or attribute value parameters, which are needed to restrict a defined graph traversal to nodes with certain properties. The path

```
path Descendent( Sex: type in PERSON ): PERSON [0:1] => Sex [0:n] =
   -Child-> & instance of Sex
end;
```

returns e.g. all children of a given person, which have the type `Sex`. The path may be applied with the actual parameter `T` = `Woman` or `T` = `Man` for `Sex` to a node of class `PERSON`; it retuns in this case a set of nodes of type `T`. Its reverse application starts at a node of type `T` = `Sex` and returns at most one `PERSON` node as its result.

The reverse application of a path is implemented very inefficiently and should be avoided whenever possible by creating an explicit reverse path declaration, as e.g.

```
path Parent : PERSON [0:n] => PERSON [0:1] = <-Child- end;
```

There is one exception from the rule that paths may but not should be traversed in reverse direction. It concerns so-called `static` (materialized) paths. These parameterless paths are evaluated once before the first attempt to traverse them; the computed result is stored in the form of additional graph edges. Any forward or backward graph traversal across materialized paths is then performed by traversing these additional edges in the appropriate direction. A variant of the incremental attribute evaluation algorithm is responsible for recomputing maybe affected materialized paths after any possibly relevant graph modification.

The graphical representation of a path declaration is a double arrow from its source to its target node type or class with its name as a label.

(130) **PathBody** ::=
```
GraphPath | OpExpr
```

A path may be defined using a textual expression-like notation (cf. rules on page 30) or a graphical notation, which offers a subset of the left-hand side elements of productions.

(131) **GraphPath** ::=
```
ApplNodeId "=>" ApplNodeId in
   [ OptLeftSideList ]
   [ OptFoldList ]
   [ OptAttConditionList ]
```

The graphical notation for the definition of a path combines the left-hand side elements of productions for the definition of graph patterns with their folding clauses and attribute conditions. It selects a source and a target node in the defined pattern as e.g. in

```
path ElderBrother: PERSON [0:n] => Man [0:n] =
   'Source => 'Target in
   begin (* defined using the textual notation *)
      obl node 'Source: PERSON; 'Parent -> 'Source: Child;
      obl node 'Parent: PERSON; 'Parent -> 'Target: Child;
      obl node 'Target: Man;
   end
   condition 'Source.Age < 'Target.Age;
end;
```

Such a path computes all matches of its graph pattern, which have a given input node as `'Source` and returns all possible `'Target` node matches as the output set.

3.24 Restriction Declarations

(132) **RestrictionDecl** ::=
[OptStaticOrVirtual] restriction DeclRestrictionId
[OptFormParLIst] ":" ApplClassId "="
RestrictionBody
end ";

A restriction declaration is the functional abstraction of a path expression, which selects a subset of its input node set. A restriction defines, therefore, a derived node set. All nodes in this set are instances of a given class or type and share a common set of properties. The syntax for the definition of restrictions is a short-hand of the syntax for the definition of path expressions. It drops the distinction between source and target nodes and their classes.

The following example of a restriction computes the set of all married PERSON nodes, i.e. the union of the set of all Woman nodes with an incoming Wife edge and the set of all Man nodes with an outgoing Wife edge:

```
restriction Married: PERSON =
       ( instance of Woman is with <-Wife- )
  or   ( instance of Man is with -Wife-> )
end;
```

It is equivalent to a path declaration with the same body, but a syntactically different signature:

```
path Married: PERSON -> PERSON = ... end;
```

The restriction syntax should be used for all those path expressions which define a subrelation of the identity relation,. i.e. select a subset of their input nodes.

A static restriction is a (parameterless) restriction, which is internally handled like a derived index attribute. The runtime system maintains an index of all nodes, which fulfill the given restriction. This index may be used to determine the set of all nodes which fulfill the corresponding restrict condition on a production's left-hand side rather efficiently. Please note that the overhead for maintaining the corresponding node index may be considerably although a variant of the incremental attribute evaluation algorithm is used for this purpose.

The graphical representation of a restriction declaration is a double arrow which points to the corresponding node class or type and has the restriction name as a label.

(133) **RestrictionBody** ::=
GraphRestriction | OpExpr

A restriction may be defined using a textual expression-like syntax or a graphical notation, which offers a subset of the left-hand side elements of productions.

(134) **GraphRestriction** ::=
ApplNodeId in
[OptLeftSideList]
[OptFoldList]
[OptAttConditionList]

The graphical definition of restrictions is very similar to the graphical definition of paths. The only difference concerns the identification of source and target nodes. Due to the fact that a restriction always returns a subset of its input nodes it is not necessary to distinguish between source and target in the graph pattern. Please note that a restriction of the form

```
restriction R( ... ): Class = 'node in ... end;
```

is a short-hand for a path declaration of the form

```
path R(...): Class => Class = 'node => 'node in ... end;
```

(135) **OptStaticOrVirtual** ::=
static | virtual

These two key words are used to distinguish "normal" or virtual path and restriction declarations from materialized or static declarations. Often needed paths should be materialized if the underlying subgraph is only changed from time to time and if the (re-)evaluation is considerably more expensive than traversing a single edge. Restrictions should not be materialized unless they are used as restrict conditions for graph pattern nodes and if possible matches for these nodes cannot be determined efficiently.

3.25 Path and Restriction Defining Expressions

(136) **OpExpr** ::=
AndOpList | ButNotOpList | ConcOpList | OrOpList
| EqualRestriction | ImpliesRestriction | IsRestriction
| OpClosureTerm

A normal path expression traverses a graph from a given set of start nodes and returns the reached set of target nodes. Each node of the input set is processed separately:

$\texttt{pathEpxr}(\{n_1, \ldots, n_k\}) = \cup_{i=1,\ldots,k}\ \texttt{pathExpr}(n_i).$

Restrictions or restrict expressions are a syntactically well-defined subset of all path expressions, which always return a subset of their input set. They do not traverse edges of the regarded graph, except for checking certain context conditions of input nodes. A restrict expression is, therefore, a path expression, which does not traverse edges or call path declarations, except inside the following four categories of restrict expressions: `EquivalentRestriction`, `ImpliesRestriction`, `IsRestriction`, and `ContextRestriction`.

(137) **OpClosureTerm** ::=
OpTerm | ClosurePlusOp | ClosureStarOp

Simple path expressions, including the transitive (reflexive) closure of simple path expressions.

(138) **OpTerm** ::=
BracketOp | ChooseOpList | LoopOpList
| ContextRestriction | TypeRestriction | ValueRestriction
| NotOp | TypeCheckOp | CardCheckOp
| HaltOp | NilOp | SelfOp
| EdgeTypeOp | RelCall | PlusRelCall | MinusRelCall

Simple path expressions, which either have a fixed number of tokens or a well-defined begin and end symbol, such as "(" and ")".

(139) **AndOpList** ::=
{ OpClosureTerm and } OpClosureTerm

The path expression

$\texttt{OpTerm}_1$ and ... and $\texttt{OpTerm}_n$

computes the intersection of the results of $\texttt{OpTerm}_1$ through $\texttt{OpTerm}_n$ for each element of a given input set separately and builds the union of the intersection results:

$(\texttt{OpTerm}_1 \text{ and } \ldots \text{ and } \texttt{OpTerm}_n)(\{n_1, \ldots, n_k\}) = \cup_{i=1,\ldots,k}\ (\texttt{OpTerm}_1(n_i) \cap \ldots \cap \texttt{OpTerm}_n(n_i)).$

(140) **ButNotOpList** ::=
{ OpClosureTerm but not } OpClosureTerm

A path expression of this kind has the following semantics:

$(\texttt{OpTerm}_1 \text{ but not } \ldots \text{ but not } \texttt{OpTerm}_n)(\{n_1, \ldots, n_k\}) = \cup_{i=1,\ldots,k}\ (\texttt{OpTerm}_1(n_i) \setminus \ldots \setminus \texttt{OpTerm}_n(n_i)).$

(141) **ConcOpList** ::=
{ OpClosureTerm & } OpClosureTerm

The "&" symbol is the standard concatenation operator for binary relations. An expression of the form

$\texttt{OpTerm}_1 \ \& \ \ldots \ \& \ \texttt{OpTerm}_n$

applies first $\texttt{OpTerm}_1$ to a given set of input nodes, then $\texttt{OpTerm}_2$ to the computed intermediate result, and so on:

$(\texttt{OpTerm}_1 \ \& \ \ldots \ \& \ \texttt{OpTerm}_2)(\text{Set}) = \texttt{OpTerm}_n\ (\ldots(\texttt{OpTerm}_1(\text{Set})\ldots)\ .$

(142) **OrOpList** ::=
{ OpClosureTerm or } OpClosureTerm

The path expression

$\texttt{OpTerm}_1$ or ... or $\texttt{OpTerm}_n$

computes the union of the results of $\texttt{OpTerm}_1$ through $\texttt{OpTerm}_n$:

$(\texttt{OpTerm}_1 \text{ or } \ldots \text{ or } \texttt{OpTerm}_n)(\text{Set}) = \texttt{OpTerm}_1(\text{Set}) \cup \ldots \cup \texttt{OpTerm}_n(\text{Set}).$

(143) **BracketOp** ::=
"(" OpExpr ")"

The syntactical construct for nesting complex path expressions inside simple path expressions.

3.26 Iterating and Conditional Path Expressions

(144) **ClosurePlusOp** ::=
`OpTerm “+”`

The + operator computes the transitive closure of a simple path expression. It recognizes cycles in the graph and avoids processing of nodes, which are already part of the result set:

`(OpTerm+)(Set) = (OpTerm & OpTerm*)(Set) .`

(145) **ClosureStarOp** ::=
`OpTerm “*”`

The * operator computes the reflexive transitive closure of a simple path expression. It recognizes cycles in the graph and avoids processing of nodes which are already part of the result set:

`(OpTerm*)(Set) = (OpTerm & OpTerm*)(Set) ∪ Set .`

(146) **ChooseOpList** ::=
`“[” { CondOp “|” } CondOp “]”`

This rather unusual construct generalizes the if-then-else-construct of other programming languages and Dijkstra’s guarded commands. Its return value is the value of the first succcessfully evaluated subexpression (from left to right). A path expression evaluation is successful if it returns at least one node applied to a regarded input node, i.e.:

$[\ OpExp_1 \mid OpExp_2 \mid \ldots\ ](n) = OpExp_1(n),$ if OpExp(n) # nil
$[\ OpExp_1 \mid OpExp_2 \mid \ldots\ ](n) = [\ OpExp_2 \mid \ldots\ ](n),$ if OpExp(n) = nil

(147) **LoopOpList** ::=
`“{” { CondOp “|” } CondOp “}”`

A loop path expression is an iterated choose path expression. Its application to a set of nodes is defined as the union of the results for each node in the input set. The loop operator executes its body as a choose path expression until all its subexpressions return the empty set. Its result set is not the set of all visited nodes (as in the case of the transitive closure), but just the set of finally reached nodes. The evaluation of the loop mantains a visited-node-set, thereby recognizing cycles in the graph and avoiding processing of already visited nodes.

```
{ Body }(n) =  n ,          if [ Body ](n) = nil
([ Body ] & { Body })(n) ,  if [ Body ](n) # nil
```

The path expression

`{ -child-> | instance of Woman & -Husband-> | instance of Man & -Wife-> }`

visits, for instance, all descendents of a given set of input nodes and all descendents of the wifes or husbands of these descendents, and ... . It returns all visited nodes (including the input set) without children and without a husband or a wife.

(148) **CondOp** ::=
`GuardOp | OpExpr`

A subexpression of a choose or a loop path expression either is a guarded path expression or a normal path expression. Its evaluation fails if it returns an empty set of nodes.

(149) **GuardOp** ::=
`OpExpr “::” OpExpr`

A guarded subexpression of a choose or loop path expression has a restriction as its guard and any path expression as its body. Its evaluation returns the empty node set if the guard fails applied to a selected input node, its evaluation returns the result of the guarded body expression otherwise. Please note that a guarded body expression, which returns the empty result set, does not trigger the selection of the following guarded subexpression (from left to right), but returns its empty set of nodes as the result of the surrounding construct.

The path expression

`{ isMarried :: -child-> | self }(n)`

returns, applied to an unmarried `PERSON` node `n`, the `PERSON` node itself. Applied to a married `PERSON` node `n` it computes first the set `S` of all its `child` nodes and continues the iteration process with

$\cup_{m \in S}$ `{ isMarried :: -child-> | self }(m) .`

This path expression returns the empty set applied to a married node without children, it returns the input node itself if applied to an unmarried node.

3.27 Node Set Restriction Expressions

(150) **EqualRestriction** ::=
OpClosureTerm "<=>" OpClosureTerm

An equ(iv)al(ent) restriction returns all those nodes of its input set for which the evaluation of the first subexpression returns the same result as the evaluation of the second subexpression:

$(OpExp_1 <=> OpExp_2)(Set) = \{ n \in Set \mid OpExp_1(n) =_{set} OpExp_2(n) \}$

The two subexpressions are both either real path expressions or restrictions. The forbidden combination of a restriction, which returns a subset of its input node set, and a path expression, which navigates to a different set of nodes in the graph, would almost always return the empty set of nodes.

(151) **ImpliesRestriction** ::=
OpClosureTerm implies OpClosureTerm

An implies restriction returns all those nodes of its input set for which the evaluation of the first subexpression returns a subset of the evaluation result of the second subexpression:

$(OpExp_1 \text{ implies } OpExp_2)(Set) = \{ n \in Set \mid OpExp_1(n) \subseteq OpExp_2(n) \}$

The two subexpressions are both either real path expressions or restrictions. The forbidden combination of a restriction, which returns a subset of its input node set, and a path expression, which navigates to a different set of nodes in the graph, would almost always return the empty set of nodes.

(152) **IsRestriction** ::=
OpClosureTerm is OpClosureTerm

An is restriction returns all those nodes of its input set, which fulfill the follwing restriction: All result nodes of the application of the first subexpression to the regarded node fulfill the second subexpression, which has to be a restriction. More precisely, this condition is defined as follows:

$(OpExp_1 \text{ is } OpExp_2)(Set) = \{ n \in Set \mid (OpExp_1 \,\&\, OpExp_2)(n) = OpExp_1(n) \}$

The is restriction may, for instance, be used to determine all PERSON nodes without brothers:

```
((<-child- & -child->) but not self) is instance of Woman .
```

The first subexpression above computes the children of the parents of a node (the parents are computed by traversing a child edge in reverse direction) without the considered node itself. The second subexpression checks that the result set of the first subexpression consists of Woman nodes only.

(153) **ContextRestriction** ::=
with OpTerm

A context restriction is valid for a given node if its subexpression applied to this node returns a non-empty set of nodes. This condition may be defined as follows:

$OpExp(Set) = \{ n \in Set \mid OpExp(n) \# \emptyset \}$.

The following expression selects, for instance, all married nodes of a given input set:

```
(instance of Woman & with -Husband->) or (instance of Man & with -Wife->).
```

(154) **TypeRestriction** ::=
instance of Factor

The type restriction selects all those input nodes, which are instances of a certain set of node types or node classes. Typical examples of type restrictions are:

```
instance of (ANIMAL but not MAMMAL)
instance of (Dog or Cat)
```

with MAMMAL being a subclass of ANIMAL
with Dog and Cat being two node types.

(155) **ValueRestriction** ::=
valid Term

The value restriction is the "gateway" from path expressions to attribute expressions. It is used to inspect node attributes in path expressions. The following path expression returns, for instance, all babies of a given set of PERSON nodes:

```
-child-> & valid( self.Age < 1 ) .
```

(156) **NotOp** ::=
not OpTerm

The not operator realizes the negation of a restriction, i.e. it returns all those nodes of the input set which do not fulfill the restriction of its subexpression. The subexpression has to be a restriction, i.e.

```
not with -child-> is a permitted path expression, but
not -child-> is not a permitted path expression.
```

3.28 Simple Path Expressions

(157) **TypeCheckOp** ::=
OpTerm ":" Type

The type checking operator should be used, whenever static analysis computes a more general type (class) for some path expression than required. It may, for instance, be used to restrict (cast) the type of the path expression

```
<-child- & not instance of Man
```

from the static type `PERSON [0:n]` (computed by the type checker) to the actual type `Woman [1:1]` in

```
path Mother: PERSON [0:n] => Woman [1:1] =
   ( <-child- & not instance of Man) : Woman [1:1]
end;
```

Please note that the type check operator is not a downcast in the sense of C or Modula-2, but checks the (node) types of its input set as follows:

```
  (OpTerm : T [x:y])(Set)
= ((OpTerm & instance of T) : [x:y])(Set)
```

It is, therefore, realized as a combination of the restriction of `Set` to all nodes of type (class) `T` followed by the application of a cardinality checking operator (see below).

(158) **CardCheckOp** ::=
OpTerm ":" Qualifier

The cardinality checking operator is useful for situations, where static type checking is not able to guarantee that the result of a certain path expression is always a well-defined or uniquely defined result. Based on the edge type definition

```
edge type child: PERSON [0:n] -> PERSON [0:n];
```

static analysis is not able to guarantee that the path expression

```
<-child- & instance of Man
```

returns at least one node or at most one node, i.e. that any `PERSON` node has at least or at most one father. The path expression

```
(<-child- & instance of Man) : [1:1]
```

has the static type `Man [1:1]`. Its application to a `PERSON` node without a father node or with more than one father node aborts and terminates the whole execution process.

(159) **HaltOp** ::= halt

The halt operator is often used as the last branch of a conditional (choose) path expression. It terminates execution immediately and converts the static type of a conditional path expression from `[0:x]` to `[1:x]`. A path expression of the form

```
[ -child-> | halt ] = -child-> : [1:n]
```

either returns a nonempty set of nodes or terminates the execution process.

(160) **NilOp** ::= nil

The nil operator always returns the empty set of nodes.

(161) **SelfOp** ::= self

The self operator returns its input node (set) without any modifications. The path expression

```
(<-child-> & -child-) but not self
```

returns, for instance, all children of the parents of a `PERSON` node, except the node itself.

3.29 Edge Traversals, Attribute Accesses, and Path Calls

(162) **EdgeTypeOp** ::=
`PlusOp | MinusOp`

The operators for forward and backward traversal of edges. They are used for the definition of edge traversing path expressions and for the definition of textual embedding clauses (cf. rules on page 27). Referenced edges are either declared as (cf. rules on page 5)

```
edge type e: SourceClass -> TargetClass;
```

or inside a node class `SourceClass` as an intrinsic attribute (cf. rules on page 7)

```
intrinsic e: TargetClass;
```

(163) **PlusOp** ::=
`"-" ApplEdgeTypeId "->"`

The plus operator traverses an edge of a given type `e` from source to target. It returns all target nodes of `e` edges which have input set nodes as sources.

(164) **MinusOp** ::=
`"<-" ApplEdgeTypeId "-"`

The minus operator traverses an edge of a given type `e` from target to source. It returns all source nodes of `e` edges which have input set nodes as targets.

(165) **RelCall** ::=
`ApplRelId [ OptActParList ]`

The identifier of the "relation call" operator is either the name of an attribute (of any kind) or the name of an edge type or the name of a path or restriction declaration. Its actual parameter list is always the empty list for attribute declarations, it corresponds to the formal parameter list of a path or restriction declaration otherwise. Typical examples of "relation calls" as the second argument of a select expression (cf. rules on page 45) are:

`n.AgeAttribute` or `n.childEdge` or `n.motherPath` or `n.marriedRestriction`.

Any expression of the form

`n.-edge->` may be abbreviated to the "relation call" `n.edge`,

any expression of the form

`n.=path=>` may be abbreviated to the "relation call" `n.path`.

(166) **PlusRelCall** ::=
`"=" ApplDAttPathId [ OptActParList ] "=>"`

A path expression of this kind is the standard notation for triggering the forward evaluation of a declared path such as (cf. rules on page 28)

```
path Parent: PERSON [0:n] => PERSON [0:n] = <-child- end;
```

or a derived attribute (cf. rules on page 8) such as

```
node class PERSON;
  derived Parent: PERSON [0:n] = self.<-child-;
end;
```

(167) **MinusRelCall** ::=
`"<=" ApplDAttPathId [ OptActParList ] "="`

A path expression of this kind offers the standard notation for the backward evaluation (reverse application) of a declared `static` path such as

```
static path Ancestor: PERSON [0:n] => PERSON [0:n] = <-child- + end;
```

or for the backward evaluation of a derived attribute (cf. rules on page 8) such as

```
node class PERSON;
  derived Ancestor: PERSON [0:n] = self.<-child- +;
end;
```

The path expression

`n.<=Ancestor=` is, therefore, equivalent to `n.-child-> +`.

Please note that non-`static` or `virtual` paths such as

```
[virtual] path Parent: PERSON [0:n] => PERSON [0:n] = <-child- end;
```

may not be traversed in reverse direction.

3.30 User Defined Functions and Formal Parameter Lists

(168) **FunctionDecl** ::=
`function` FuncOpName “:” [OptFormParList] “->” Type “=”
Expression
`end` “;”

A function declaration is either a normal function declaration with an arbitrary number of in-parameters or a operator declaration. Operators have either two in-parameters and are called in infix notation, or they have one parameter and are then called in prefix notation. Please note that functions do not have out-parameters although the context-free syntax allows their definition.

(169) **FuncOpName** ::=
DeclFunctionId | DeclOpId

The name of a normal function (`DeclFunctionId`) is a sequence of alphanumeric characters. The name of an operator (`DeclOpId`) is either a sequence of nonalphanumeric characters or any sequence of characters which has “‘” as the leading delimiter and “‘” as the trailing delimiter. Legal operator names are e.g.:

`‘conc’` and `!` and `++` and `$` and `~` and `‘$’` and `‘121’`

but not

`conc` and `‘c` and `121`.

(170) **OptFormParList** ::=
“(” { ParDecl “;” } ParDecl “)”

A formal parameter list of functions, productions, etc. is a list of in- and out-parameter declarations. The order of parameter declarations in the list determines the required order of actual parameter values of function calls, production calls, etc.

(171) **ParDecl** ::=
InParDecl | OutParDecl

A formal parameter is either an in-parameter or an out-parameter. In-parameters are read-only, out-parameters are write-only for the body of the declaration, which has the formal parameter list. There are no in-out-parameters or call-by-reference-parameters as in many other languages.

(172) **InParDecl** ::=
InParIdList “:” Type

An in-parameter declaration is a list of parameter identifiers followed by their common type definition. A declaration of the form

`a, b : T`

is just an abbreviation for

`a : T; b : T .`

(173) **InParIdList** ::=
{ DeclInParId “,” } DeclInParId

This is a comma-separated list of formal in-parameters.

(174) **OutParDecl** ::=
`out` OutParIdList “:” Type

An out-parameter declaration is a list of parameter identifiers followed by their common type definition. Out-parameter declarations with more than one identifier are expanded in the same way as in-parameter declarations with more than one identifier. An out-parameter declaration of the form

`out P1, P2 : Type`

is, therefore, an abbreviation for

`out P1 : Type; P2 : Type .`

(175) **OutParIdList** ::=
{ DeclOutParId “,” } DeclOutParId

This is a comma-separated list of formal out-parameters.

3.31 Attribute Value and Node Set Computing Expressions

(176) **Expression** ::=
`Term | InfixOpExpr | IsExpr`

An expression is either a simple term or the call of a binary operator in infix notation or it is an `is` expression, which serves as a kind of gateway from attribute expressions to path expressions.

(177) **InfixOpExpr** ::=
`Term { InfixOpId Term } InfixOpId Term`

This is an expression of the form
$\texttt{Term}_0 \texttt{op}_1 \ldots \texttt{op}_n \texttt{Term}_n$.
Terms and operators are evaluated from left to right without taking any operator precedences into account.

(178) **InfixOpId** ::=
`BinaryStandardOp | ApplOpId`

A binary operator called in infix notation is either one of the standard operators or a user-defined operator.

(179) **BinaryStandardOp** ::=
```
    Or | And | ButNot | Implies | Equivalent | In
| Equal | Unequal | Greater | Less | GrEqual | LeEqual
| Plus | Minus | Mult | Div | Mod | Concat
```

These are all built-in operators with two arguments for standard data types.

(180) **Term** ::=
`Factor | PrefixOpExpr`

A term is either a simple factor or the call of a unary operator in prefix notation.

(181) **PrefixOpExpr** ::= `PrefixOpId Factor`

The call of a unary operator in prefix notation. Unary operators have higher precedence than binary operators, i.e. `1 + -2` is evaluated as `1 + (-2)`.

(182) **PrefixOpId** ::=
`UnaryStandardOp | ApplOpId`

A unary operator called in prefix notation is either a standard operator or a user-defined operator.

(183) **UnaryStandardOp** ::=
```
    Minus | Not | All | Char | Ord
```

The first two operators are built-in operators with one argument for standard types. The `all` operator is used for applying binary operators to sets of arguments of any type.

(184) **Factor** ::=
```
    ConstExpr | FuncVarParExpr
| StringToValueExpr | ValueToStringExpr
| ExistExpr | ForAllExpr | UseExpr
| BracketExpr | ChooseExprList
| ApplNodeId | ApplNewNodeId
| TypeCheckExpr | CardCheckExpr
| SelectExpr | TypeOfExpr | InstanceOfExpr
```

All offered kinds of factors. Most of them will be explained later on.

(185) **ConstExpr** ::=
```
    Halt | False | True
| Nil | Number | StringConst
| Self
```

These are all built-in constants of type `integer`, `string`, `boolean`, and so on, except the operator `self` which is needed for referencing an attribute's node inside its own evaluation rule.

(186) **Halt** ::= `halt`

The operator `halt` is mainly used as the last branch of a choose expression. Its evaluation causes immediate termination of a running execution process.

3.32 Function Calls and Actual Parameter Lists

(187) **FuncVarParExpr** ::=
`FuncVarPar [ OptActParList ]`

Expressions of this category are either function calls with an optional parameter list or applied occurrences of various kinds of identifiers without a parameter list.

Variable identifiers and in-parameters denote their values, node type identifiers return themselves as value, and node class identifiers are evaluated to the set of all their node types (directly or indirectly derived from the given node class). The occurrence of a node class `C_1` with subtype `T_1` and subclass `C_2`, which has in turn a subtype `T_2`, is for instance evaluated to the set `{T_1,T2}`.

Standard or user-defined function calls are evaluated as usual. They have the form

$f(exp_1, \ldots , exp_n)$

for functions with at least one argument and with exp_i as actual parameter values or the form

`f`

for functions without any formal parameters at all.

(188) **FuncVarPar** ::=
`StandardFunc | ApplFuncVarParId`

The identifier of a standard function or a user-defined function or the identifier of a variable, in-parameter, node type, or node class.

(189) **StandardFunc** ::=
`Card | Empty | Elem | SetOf`
`| Length | Substr`

The standard functions for strings and sets of any element type.

(190) **OptActParList** ::=
`“(” { ActPar “,” ActPar } “)”`

A list of actual parameter values for functions and other kinds of functional abstractions. Formal parameter lists are explained together with function declarations (cf. rules on page 35).

(191) **ActPar** ::=
`Expression | ActOutPar`

Formal parameters are either in- or out-parameters in the general case. Actual values for in-parameters are expressions, actual values for out-parameters are identifiers. The same identifier may not appear as actual value for two out-parameters of the same actual parameter list. Please note that functions have in-parameters and return values, but no out-parameters (out-parameters are used later on for calls of graph rewrite rules etc.).

(192) **ActOutPar** ::=
`out ApplVarOutParId`

The actual value of an out-parameter is the identifier of a variable or another out-parameter.

3.33 Standard Types Boolean and Sets of Any Type

(193) **True** ::= true

The boolean constant true (internally realized as integer value 1).

(194) **False** ::= false

The boolean constant false (internally realized as integer value 0).

(195) **Not** ::= not

The operator not builds the complement of boolean values. It is the only boolean operator which is not overloaded. All following boolean operators are also used as set operators.

(196) **Or** ::= or

Computes the logical “or” of two boolean values or the union of two sets of arbitrary types. Single elements are automatically converted into singleton sets. Sets may be constructed as follows:

```
{1,2,3} = 1 or {2,3} = 1 or 2 or 3 .
```

Please note that sets of boolean values are not supported. It is, therefore, always clear from context whether or denotes the logical $\vee$ or the set union operator $\cup$.

(197) **And** ::= and

Computes the logical “and” of two boolean values or the intersection of two sets of arbitrary types. The discrimination between the usage as the boolean $\wedge$ operator or the usage as the set intersection operator $\cap$ is as simple as in the case of or above.

(198) **ButNot** ::= but not

Computes the difference of two sets of values if applied to non-boolean arguments. Its behavior for boolean values is as follows:

```
a but not b := a and (not b)
```

(199) **Implies** ::= implies

Compares two sets of values and returns true (false) if the first set is (not) a subset of the second set. Its behavior for boolean values is as follows:

```
a implies b := (not a) or b .
```

(200) **Equivalent** ::= “<=>”

Compares two sets of values and returns true (false) if these sets are (not) equal. Its behavior for boolean values is as follows:

```
a <=> b := (a and b) or (not a and not b) .
```

(201) **Nil** ::= nil

The operator nil represents an undefined value or an empty set of values of any possible type.

(202) **Elem** ::= elem

The operator selects one randomly chosen element from a set of elements. Its usage in assignments of the form

```
ElementVar := elem( SetExpression )
```

is recommended, but not enforced (missing set-to-element conversions are automatically inserted). Its execution fails if the argument is the empty set. It depends on the context of the built expression whether this kind of failure causes the execution to terminate or triggers backtracking.

(203) **SetOf** ::= set of

The operator offers the inverse function to elem. It takes a single element as input and returns a singleton set. The usage of this operator is again optional; needed element-to-set conversions are automatically inserted, whenever needed.

(204) **Card** ::= card

The expression card(set) returns the size of its parameter set as a nonnegative integer value.

(205) **Empty** ::= empty

The boolean test for empty sets of any possible type.

(206) **In** ::= in

The expression e in S returns true (false) if element e is (not) in set S.

3.34 Relational Operators

(207) **Equal** ::= “=”

The operator compares two elements which have a common type or superclass. It returns `true` if two compared attribute values are equal or if two compared node objects are identical.

(208) **Unequal** ::= “#”

This is the negation of “=”, i.e.

```
a # b := not (a = b) .
```

(209) **Greater** ::= “>”

The operator compares two `integer` values (but not sets of `integer` values) as usual.

(210) **Less** ::= “<”

The operator compares two `integer` values (but not sets of `integer` values) as usual.

(211) **GrEqual** ::= “>=”

The operator compares two `integer` values (but not sets of `integer` values) as usual.

(212) **LeEqual** ::= “<=”

The operator compares two `integer` values (but not sets of `integer` values) as usual.

3.35 Standard Types Integer and Integer Set

(213) **Number** ::= ...

An integer constant, called number, is a sequence of digits ∈ `0` | ... | `9`, which may be stored in four bytes.

(214) **Plus** ::= “+”

The expression `a + b` computes the sum of two integer elements or two sets of integer elements.

```
{1, 2} + {3,4} = {4,5,6}
```

returns a set of integer values, where each element of the first set is added to each element of the second set.

(215) **Minus** ::= “-”

The expression `a - b` computes the difference of two integer elements or two sets of integer elements in the same manner as + computes the sum of (two sets of) integer values:

```
{4,5} - {3,8} = {1,2,-4,-3} .
```

The expression `-a` returns all elements of `a` with changed sign, i.e.

```
- {4,5} = {-4,-5} .
```

(216) **Mult** ::= “*”

The expression `a * b` multiplies (two sets of) integer values.

(217) **Div** ::= “/”

The expression `a / b` divides (all elements of) `a` by (all elements of) `b`, neglecting the remainder.

(218) **Mod** ::= “%”

The expression `a % b` computes the remainder of `a / b`. This is again a set in the general case, e.g.:

```
{1,4,-7} % {2,3} = {1,0,-1} .
```

(219) **Char** ::= `char`

The operator converts a character (`string` of length 1) into its ordinal number. Applied to a set of characters it computes the set of their ordinal numbers, as e.g. in:

```
(char("c","e")-char("a")) = {2,4} .
```

(220) **Ord** ::= `ord`

The operator `ord` is the inverse function of the operator `char`. It translates any number between `0` and `255` into the appropriate ASCII character (`string` of length 1).

3.36 Standard Types String and String Set

(221) **StringConst** ::= ...

A string is a sequence of arbitrary characters (except double quotes) with a leading and a trailing double quote “"”. Masking of double quotes inside strings is not (yet) supported. The length of a string is restricted to 250 characters. Examples of legal strings are:

```
"abcd''"    and "$^&%#@^*" .
```

(222) **Concat** ::= “&”

The operator takes two (sets of) strings as input and concatenates them pairwise, as e.g. in:

```
  {"Andy","Hannah","Margot"} & " " & "Schürr"
= {"Andy Schürr","Hannah Schürr","Margot Schürr"}.
```

(223) **StringToValueExpr** ::=
value “(” Expression “,” Type “)”

The operator converts (a set of) strings into expressions of a given type. The conversion process is split in an element-wise conversion of strings to values, followed by a restriction of the computed set of values to the given `Type` expression, i.e.

```
value( {"a", "-2"}, Type ) = {"a",-2} : Type .
```

It depends on the specific form of `Type` whether failing string conversions return the empty set of values or stop the overall execution process. The expression

```
value( "abc", integer ) = value( "abc", integer [1:1] )
```

is for instance expected to return a single well-defined integer value, but its first argument is not the `string` representation of an `integer`. As a consequence its evaluation fails and terminates the overall execution process. The expression

```
value( "abc", integer [0:1] )
```

on the other hand simply returns the permitted undefined `integer` value `nil`.

Please note that the operator works for any built-in attribute type as well as for imported types, node identifiers, and node types. Assuming that `n_i` is a node of type `T_i`, `T_i` is a node type derived from class `C_i` but not from class `C_j`, the following expressions are evaluated as follows:

```
value( {"n_1", "n_2"}, T_1 [0:n] ) = {n_1}
value( {"T_1", "T_2"}, type in C_1 [0:n] ) = {T_1} .
```

(224) **ValueToString** ::=
string “(” Expression “)”

The operator performs the inverse computation of the `value` operator, i.e. translates any given set of values (of the same type) into a set of strings:

```
string( {123,456} ) = {"123","456"} )
string( "abc" ) = "abc" .
```

Please note that the operator works for any built-in attribute type as well as for imported types, node identifiers, and node types.

(225) **Length** ::= length

The operator returns the length of (a set of) strings as a nonnegative integer value:

```
length( {"","abc","d"} ) = {0,3,1} .
```

(226) **Substr** ::= substr

The operator selects substrings of a given string. It needs three arguments for this purpose:

```
substr( str, fpos, lpos  ).
```

The three arguments are used as follows:
(1) `str` is the regarded string,
(2) `fpos` is the first position of the selected substring, and
(3) `lpos` is the last position of the selected substring.
The function returns the empty string "" for `fpos > lpos`. Its parameter `fpos` is set to `1` if it is less then `1`, its parameter `lpos` is set to `length(str)` if it is greater than `length(str)`.
All three arguments may be sets of values:

```
  substr( {"abcd","efg"}, {1,2}, {2,4} )
= {substr( "abcd", 1, 2 ), ... , substr( "efg", 2, 4 )}
= {"ab","ef","b","f","abc","efg","bcd","fg"} ) .
```

3.37 Expression Iterators

(227) **ExistExpr** ::=
```
exist VarDeclList "::"
  Expression
end
```
This is the existential quantifier of first order predicate logic. An expression of the form
```
exist a := elem( {1,3} );
      b := elem( {1,3} );
      c := elem( {2,3} ) ::
  a < c < b
end
```
executes the boolean subexpression between :: and end for any possible combination of value assignments to variables a, b, and c, respectively. It returns true if and only if any subexpression evaluation returns true. The evaluation of the example above yields true, because of
```
(a = 1) < (c = 2) < (b = 3) .
```

(228) **ForAllExpr** ::=
```
for all VarDeclList "::"
  Expression
end
```
This is the all quantifier of first order predicate logic. It has about the same behavior as the existential quantifier, except the fact that its result is true if and only if all its subexpressions are evaluated to true.

(229) **UseExpr** ::=
```
use VarDeclList "::"
  Expression
end
```
The use expression allows the assignment of subexpression values to local variables. These variables may be referenced in the subexpression between :: and end. Please note that it is possible to assign a set of elements to a local variable which may only hold a single element:
```
use v : integer := elem( {1,2,3} ) ::
  v + v
end
```
In this case, the subexpression between :: and end is evaluated for any possible assignment of a single value to the declared variable. The result of the whole expression is the set of all possible subexpression evaluations. The result of evaluating the example above is
```
{(1 + 1),(2 + 2),(3 + 3)} = {2,4,6} .
```

(230) **All** ::= all

The all operator transforms the application of a binary operator or function with two parameters to a single start element on one side and a set of elements on the other side into a nested sequence of operator applications. Each operator or function call takes one element of the given set as one input value and the until now computed value as the other input value:
```
0 + all {1,2,3, ... } = ( ... ((0 + 1) + 2) + 3) + ...
max(all {1,2}, 0) = max(2, max(1,0))
max( all nil, 0 ) = max( all {}, 0 } = 0 .
```
The performed nesting of operator or function calls requires that the type (class) of the involved actual set-parameter is the same type (class) as the function's return type (class) or that it is a subtype (subclass) of the function's return type (class). Furthermore, the regarded operator or function should be associative and commutative such that the order of element selections in the given set has no influence on the computed result.

3.38 Variable Declarations and Type Definitions

(231) **VarDeclList** ::=
```
{ VarDecl ";" }
  VarDecl
```
This is the declaration of a list of local variables. The order of list elements is of no importance. The expressions which assign initial values to introduced variables may not access other variables in the same list, but they may access variables or parameters defined in the surrounding block:
```
a : integer := x;
b : integer := a
```
are valid variable declarations if x and a are integer variables of the surrounding block; variable b does not receive the value of the just defined a := x, but the value of a previously declared variable a of a surrounding block as initial value.

(232) **VarDecl** ::=
```
VarIdList [ ":" Type ] [ ":=" Expression ]
```
A variable declaration is a list of variable identifiers followed by their common type definition and their common initial value definition. The initial value definition may be omitted if the variable's value is defined later on, a variable's type definition may be omitted if its type may be inferred from a given initial value. The declaration
```
a, b := {1,3}
```
is e.g. expanded to
```
a : integer [1:n] := {1,3};
b : integer [1:n] := {1,3} .
```
A variable declaration of the form
```
use a, b : integer [1:1] : = elem( IntSet ) ::
  exp
end
```
is prohibited. It is not clear whether it should be expanded to
```
use a : integer [1:1] : = elem( IntSet ) ::
  use b : integer [1:1] := a ::
    exp
  end
end
```
or to
```
use  a : integer [1:1] : = elem( IntSet );
     b : integer [1:1] : = elem( IntSet ) ::
  exp
end
```
i.e. whether variables a and b always have the same value taken from IntSet or whether they receive values from this set independently from each other.

(233) **VarIdList** ::=
```
{ DeclVarId "," } DeclVarId
```
A list of local variable identifiers. The order of list elements is of no importance.

3.39 Type and Meta Type Definitions

(234) **Type** ::=
```
QualifiedType | BooleanType
```
A variable type (parameter type, attribute type, etc.) is either a qualified type or a `boolean` type. Qualified types are those types for which sets of values may be constructed. Sets of `boolean` values are not supported in order to avoid the introduction of a four-valued logic with {} = “unknown” and {`true`,`false`} = “maybe”.

(235) **QualifiedType** ::=
```
SimpleType [ Qualifier ]
```
A qualified type is any type identifier, except `boolean`, followed by an optional qualifier. The default value for the qualifier is `[1:1]`. The type definitions have to be read as follows:
- `T [0:1]` defines a partially defined variable, ... of type `T` (value is element of type `T` or `nil`).
- `T [1:1]` defines an always defined element-valued variable, parameter etc. of type `T`.
- `T [1:n]` defines a variable, parameter etc. which has a nonempty `T`-set as value.
- `T [0:n]` defines a variable, parameter etc. which has a maybe empty `T`-set as value.

(236) **SimpleType** ::=
```
  ApplTypeId
| MetaType
| IntegerType
| StringType
```
Identifiers of imported attribute types, node types, node classes, type-containing variables, and type-containing in-parameters are permitted type identifiers. Attribute type identifiers are used to define attribute value containing variables etc.. Node type and node class identifiers are used to define node reference containing variables. Please note that a node class identifier is a short-hand for the union of all node types belonging to this class. The variable declaration
```
v : Class [0:n];
```
defines for instance a container for sets of nodes of any node type directly or indirectly derived from node class `Class`.

Variables or in-parameters may have a so-called meta type as their type. They do not have attribute values or node references, but node types as their values. A meta type is the type of a type value. The identifiers of these variables or parameters may be used as type identifiers. They introduce parametric polymorphism:
```
function f : ( Type : type in C; a, b : Type [0:n] ) -> Type [0:n] =
  ...
end;
```
The function `f` takes a node type `T` of class `C` as first actual in-parameter and two sets of node references of the actual node type `T` as second and third in-parameter. It returns a set of node references of the actual node type `T` as its result.

The really simple types are the built-in standard types `integer` and `string`.

(237) **MetaType**
```
type in ApplNodeClassId
```
A meta type is the type of a type value. Types are almost first-order objects, which have their own meta attributes, are legal values of variables, and may be used as actual parameters of functions, productions etc.
```
v : type in C := T
```
is for instance a variable, which has a node type `T` derived from class `C` as initial value.

(238) **IntegerType** ::= `integer`

One of the three predefined standard types for attribute, variable, ... values.

(239) **StringType** ::= `string`

One of the three predefined standard types for attribute, variable, ... values.

(240) **BooleanType** ::= `boolean`

One of the three predefined standard types for attribute, variable, ... values.

3.40 Brackets and Conditional Expressions

(241) **BracketExpr** ::=
“(” Expression “)”

Brackets may be used to enforce the evaluation of binary operators in a certain order (binary operators have no associated precedence laws) as e.g. in

```
a + (b * c)
```

or to switch from Term expressions back to general Expressions.

(242) **ChooseExprList** ::=
“[” { CondExpr “|” } CondExpr “]”

This rather unusual construct generalizes the if-then-else-construct of other programming languages and Dijkstra’s guarded commands. Its return value is the value of the first succcessfully evaluated subexpression (from left to right):

```
[ a > b :: a | b ]
```

returns the maximum of a and b,

```
[ Set1 | Default ] = [ not empty( Set1 ) :: Set1 | Default ]
```

returns Set1 if it is not the empty set, it returns the Default value otherwise.

(243) **CondExpr** ::=
GuardExpr | Expression

A conditional expression of a choose expression is a guarded expression with an explicit boolean condition or an unguadred expression with an implicit boolean condition. The implicit boolean condition in the latter case requires that the evaluation of the given expression yields at least one result, i.e. is unequal to the empty set (the undefined value nil).

(244) **GuardExpr** ::=
Expression “::” Expression

A guarded expression is a boolean expression followed by an expression of any type. The boolean expression is the guard for the following expression. It determines exclusively, whether or not its choose expression branch is selected. The expression

```
[ true :: nil | ... ]
```

returns e.g. always the undefined value due to the fact that the guard of its first branch is always true.

(245) **IsExpr** ::=
Term is OpTerm (

A boolean expression which returns true if and only if each node determined by Term fulfills the restriction OpTerm. Important examples are

```
Node is instance of (Class1 or Class2),
```

which checks whether node Node belongs to class Class1 or to class Class2, or

```
Node is with (-E1-> or <-E2-)
```

which checks whether node Node is the source of an E1 edge or the target of an E2 edge, or

```
Node is okay,
```

where okay is either the name of a restriction or the name of a boolean attribute.

3.41 Attribute, Node, and Node Type Selections

(246) **Self** ::= self

References the regarded node inside intrinisic or derived attribute declarations, constraint declarations and redefinitions of these attributes and constraints. It may also be used as a reference to the regarded node type inside meta attribute declarations and redefinitions.

(247) **ApplNodeId** ::= ...

The applied occurrence of the identifier of a production's left-hand side node. It has "'" as a leading character. It may be used to reference a matched node (set of nodes) or the old attribute values of this node (node set) . The return clause

```
return OutPar := '1.Att;
```

assigns e.g. the old attribute value of node '1 (the value before the execution of any required graph modifications) to the out-parameter OutPar.

(248) **ApplNewNodeId** ::= ...

The applied occurrence of the identifier of a production's right-hand side node. It has "'" as a trailing character. It may be used to reference a matched preserved or a new node (set of nodes) or the new attribute values of this node (node set) . The return clause

```
return OutPar := 1'.Att;
```

assigns e.g. the new attribute value of node 1' (the value after the execution of all required graph modifications) to the out-parameter OutPar.

(249) **SelectExpr** ::=
Factor "." OpTerm

The select construct applies a path expression OpTerm to the result of an expression Factor. Important special cases of path expressions are

```
Node.Attr  and  NodeType.MetaAttr ,
```

which are used to access an intrinsic or derived attribute Attr of a node Node or a meta attribute MetaAttr of a node type NodeType. The result of Factor may be a set of nodes or node types in the general case. An expression of the form

```
Node.MetaAttr
```

is automatically expanded to

```
Node.type.MetaAttr .
```

(250) **TypeOfExpr** ::=
Factor "." type

Applied to a single node or a set of nodes it returns the type of this node or the set of types of these nodes. The type operator may not be applied to node types (classes) or normal attribute values like integer or string.

(251) **InstanceOfExpr** ::=
instance of Factor

Applied to a single node type or a set of node types or a node class it returns the set of all nodes of this type or class in the regarded graph. The expression

```
card( instance of (Class or Type) )
```

returns, for instance, the number of all nodes of a node type which is either the node type Type or which is derived from node class Class.

3.42 Type and Cardinality Constraint Checking Expressions

(252) **TypeCheckExpr** ::=
`Factor ":" Type`

The type checking operator should be used, whenever static analysis computes a more general type (class) for a (sub-)expression than required. It may, for instance, be used to restrict (cast) the type of the expression

```
P.<-child-.(not instance of Man)
```

from the static type `PERSON [0:n]` (computed by the type checker) to the actual type `Woman [1:1]` in

```
function Mother: (P: PERSON [1:1]) -> Woman [1:1] =
  (P.<-child-.(not instance of Man)) : Woman [1:1];
end;
```

Please note that the type check operator is not a downcast in the sense of C or Modula-2, but checks the (node) types of its input set as follows:

```
  Factor : T [x:y]
= (Factor.instance of T) : [x:y]
```

It is, therefore, realized as a combination of the restriction of `Set` to all nodes of type (class) `T` followed by the application of a cardinality checking operator (see below).

(253) **CardCheckExpr** ::= `Factor ":" Qualifier`

The cardinality checking operator is useful for situations, where static type checking is not able to guarantee that the result of a certain expression always delivers a well-defined or uniquely defined result. Based on the edge type definition

```
edge type child: PERSON [0:n] -> PERSON [0:n];
```

static analysis is not able to guarantee that the expression

```
self.(<-child- & instance of Man)
```

returns at least one node or at most one node, i.e. that any `PERSON` node has at least or at most one father. The expression

```
self.(<-child- & instance of Man) : [1:1]
```

has the static type `Man [1:1]`. Its application to a `PERSON` node without a father node or with more than one father node aborts and terminates the whole execution process.

3.43 Cardinality Qualifiers for Relation Types, Attribute Types, etc.

(254) **Qualifier** ::= `NullToOne | One | OneToMany | NullToMany`

All possible cardinality constraints for edge types, attributes, variables, ...

(255) **NullToOne** ::= `"[0:1]"`

Indicates or requires the existence of at most one ... .

(256) **One** ::= `"[1:1]"`

Indicates or requires the existence of exactly one ... .

(257) **OneToMany** ::= `"[1:n]"`

Indicates or requires the existence of at least one ... .

(258) **NullToMany** ::= `"[0:n]"`

Indicates or requires the existence of an arbitrary number of ... .

4 References

[AE94] Andries M., Engels G.: *Syntax and Semantics of Hybrid Database Languages*. In Schneider H. J., Ehrig H. (eds.): Graph Transformations in Computer Science, volume 776 of Lecture Notes in Computer Science, pp. 19–36, 1994.

[HKW98] Heimann P., Krapp C.-A., Westfechtel B.: *Graph-Based Software Process Management*. Int. Journal of Software Engineering and Knowledge Engineering: Special Issue on Graph Grammar-based Specification, 1998. to appear.

[KS97] Klein P., Schürr A.: *Constructing SDEs with the IPSEN Meta Environment*. In Proc. 8th Conf. on Software Engineering Environments (SEE'97), pp. 2–10. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[Nag96] Nagl M. (ed.): *Building Thightly Integrated (Software) Development Environments: The IPSEN Approach*, volume 1170 of Lecture Notes in Computer Science. Springer Verlag, Berlin, 1996.

[Roz97] Rozenberg G. (ed.): *Handbook on Graph Grammars: Foundations*, volume 1. World Scientific, Singapore, 1997.

[RS97] Rekers J., Schürr A.: *Defining and Parsing Visual Languages with Layered Graph Grammars*. Journal of Visual Languages and Computing, 8(1):27–55, 1997.

[Sch91] Schürr A.: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen Anwendungsbeispiele und Werkzeugunterstützung*. Deutscher Universitäts-Verlag, Wiesbaden, 1991.

[Sch97] Schürr A.: *Programmed Graph Replacement Systems*. In chapter 7 in [Roz97], pp. 479–546, 1997.

[SWZ95a] Schürr A., Winter A., Zündorf A.: *Visual Programming with Graph Rewriting Systems*. In *Proc. IEEE Symposium on Visual Languages (VL'95)*, pp. 195–202. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[SWZ95b] Schürr A., Winter A. J., Zündorf A.: *Graph Grammar Engineering with PROGRES*. In Schäfer W., Botella P. (eds.): Proc. 5th European Software Engineering Conf. (ESEC'95), volume 989 of Lecture Notes in Computer Science, pp. 219–234. Springer Verlag, Berlin, 1995.

[SWZ95c] Schürr A., Winter A. J., Zündorf A.: *Spezifikation und Prototyping graphbasierter Systeme*. In Proc. Fachtagung Software-Technik'95, pp. 86–97. GI SofTech NRW, 1995.

[SWZ96] Schürr A., Winter A. J., Zündorf A.: *Spezifikation und Prototyping graphbasierter Systeme*. Informatik – Forschung und Entwicklung, 11(4):191–202, 1996.

[Wes91] Westfechtel B.: *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*, volume 280 of Informatik-Fachberichte. Springer-Verlag, Berlin, 1991.

[Zün96] Zündorf A.: *Eine Entwicklungsumgebung für programmierte Graphersetzungssysteme*. Deutscher Universitätsverlag, Wiesbaden, 1996. Dissertation, RWTH Aachen.

5 Index