Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

Andy Schürr[‡]

Lehrstuhl für Informatik III
Aachen University of Technology
Ahornstraße 55, D-5100 Aachen,
West Germany

Abstract: The language **PROGRESS** presented within this paper is the first **strongly typed** language which is based on the concepts of **PRO**grammed Graph **RE**writing **S**y**S**tems. This language supports a **data flow oriented** style of programming (by means of attribute equations), an **object oriented** style of programming (by supporting multiple inheritance and dynamic bind of attribute designators to their value defining equations), a **rule based** style of programming (by using graph rewrite rules), and an **imperative** style of programming (by composing single graph rewrite rules to complex transformation programs). Both the language and its underlying formalism are based on an experience of about seven years with a **model oriented** approach to the specification of document classes and document processing tools (of the Integrated Programming Support ENviroment **IPSEN**). This approach, called **graph grammar engineering,** is characterized by using **attributed graphs** to model object structures. Programmed graph rewriting systems are used to specify operations in terms of their effect on these graph models. This paper informally introduces PROGRESS' underlying graph grammar formalism and demonstrates its systematic use by specifying parts of a desk calculator's functional behaviour.

1. Introduction

Modern software systems for application areas like office automation, and software engineering are usually highly interactive and deal with complex structured objects. The systematic development of these systems requires precise and readable descriptions of their desired behaviour. Therefore, many specification languages and methods have been introduced to produce formal descriptions of various aspects of a software system, such as the design of object structures, the effect of operations on objects, or the synchronization of concurrently executed tasks. Many of these languages use **special classes of graphs** as their underlying data models. Conceptual graphs /So 84/, (semantic) data base models /Me 82/, petri nets /GJ 82/, or attributed trees /Re 84/ are well-known examples of this kind.

[‡] Supported by Stiftung Volkswagenwerk

Within the research project **IPSEN** (an acronym for Integrated/Incremental Programming Support ENvironment) a **graph grammar based specification method** has been used to model the internal structure of software documents and to produce executable specifications of corresponding document processing tools, as e.g. syntax-directed editors, static analyzers, or incremental compilers /ES 85/. The development of such a specification, which is termed **'programmed graph rewriting system'** consists of two closely related subtasks. The first one is to design a graph model for the corresponding complex object structure. The second one is to program object (graph) analyzing and modifying operations by composing sequences of sub-graph tests and graph rewrite rules.

Based on experiences of about seven years with this IPSEN specific approach to the **formal specification and systematic development** of software, we were able to adapt the original formalism (introduced in /Na 79/) to the requirements of this application area (cf. /En 86, Le 88a/). Furthermore, a method, called **graph grammar engineering**, has been developed for the construction of large rewriting systems in a systematic engineering-like manner (cf. /ES 85, EL 87/). Parallel to the continuous evolution of the graph grammar formalism and the graph grammar engineering method, the design of a **graph grammar specification language** is in progress. A first version of this language, termed **PROGRESS** (for **PRO**grammed Graph **RE**writing System Specification), was fixed a few months ago, and a **prototype of a programming environment** for this version of the language is under development.

The purpose of this paper is to survey the language PROGRESS. It is addressed to those readers who are familiar with the formalism of attribute (tree) grammars and tree rewriting systems (cf. /Re 84, MW 84/) but not with the formalism of (attribute) graph grammars and graph rewriting systems. Therefore, the next section is dedicated to an informal introduction of our **graph grammar formalism.** This section also introduces the running example which is used throughout the whole paper for demonstration purposes. Section 3 presents a typical cut out of the features of the **language PROGRESS** and demonstrates the supported way of 'programming'. Section 4 contains some remarks about **related work** and the last section discusses **future development plans** for this graph grammar based specification language.

2. The Graph Grammar Formalism

Writing about PROGRESS one has to explain and discuss topics at least at two different levels. The first one is that of the language's underlying **graph grammar formalism** which builds a fundament for the semantic definition of the language. The second one is that of **language design issues** and comprises tasks like defining the abstract syntax, the name binding rules, and the concrete representations for the new language. To avoid a confusion of these two levels within the paper, I decided to dedicate this section completely to the informal introduction of our graph grammar formalism and to defer the presentation of the language itself to the following section.

The first subsection introduces the formalism's underlying data model - **attributed, node and edge labeled, directed graphs** - and demonstrates the mapping of complex object structures onto this kind of graphs. The second subsection sketches the specification of the functional behaviour of object (graph) processing tools by means of **graph rewrite rules**. Within this section and for the remainder of the paper I use a subset of the well-known applicative programming language 'Exp' as a running example (variants of this example may be found in /RT 84, JF 84/).

2.1 Data Modeling with Attributed Graphs

This subsection introduces the class of **attributed, node and edge labeled, directed graphs** (in the sequel just called graphs). Based on the afore-mentioned 'Exp' language I will discuss how to represent the sentences of a certain programming language by the instances of a certain subclass of graphs.

Therefore, I start our explanations with the description of the language 'Exp'. This language characterizes all legal input sequences for a very primitive desk calculator. Landin's 'let' construct /La 66/ is used to name and reuse intermediate computation results. The usual scoping rules for block structured programming languages direct the binding of applied occurrences of names to their corresponding 'let'-definitions. Figure 1 defines the abstract syntax of the language in a manner of writing called the **operator / phylum** notation (cf. /No 87/). The somewhat artificial subphyla 'NIL__EXP', 'BINARY_EXP' etc. have been introduced to group operators with same properties and to emphasize the similarities between the abstract syntax description of a language and the PROGRESS declarations for a corresponding graph scheme (cf. section 3.).

```
EXP ::= NIL_EXP | BINARY_EXP | CONST_EXP | DEF_EXP | NAME_EXP
          (* 'NIL_EXP' etc. are subphyla of the phylum 'EXP'. *)

NAME ::= NameDef ( String )        (* Defining occurrence of an identifier. *)

NAME_EXP ::= NameUse ( String ) (* Applied  occurrence of an identifier. *)

DEF_EXP ::= Let ( NAME, EXP, EXP )(* Binds the first expression to applied occurrences
                                     of the name within the second expression. *)

BINARY_EXP ::= Prod ( EXP, EXP ) | Quot ( EXP , EXP ) | . . .

CONST_EXP ::= DecConst ( String )

NIL_EXP ::= NilExp ()              (* Placeholder for unexpanded subexpressions. *)
```

Figure 1 : Abstract syntax of the language 'Exp'.

Figure 2 presents the **text and graph representation** of a typical sentence of the language 'Exp'. This sentence still contains one unexpanded subexpression with an undefined value represented by '(<EXP>)'. As a consequence, the values of the two (sub-) expressions starting with 'let x is ...' are undefined, too. The computation of the subexpression 'let y is 2 in x*y' yields the value '16' due to the fact that 'x' is bound to '8' (by the outermost 'let') and y to '2'.

Starting from the definition of the abstract syntax in figure 1, the **systematic development** of the graph representation has been directed by the following guide-lines (cf. also /En 86, EL 87/):

let x is 8 in ⌊let x is ⌊let y is 2 in x * y⌉ in (<EXP>) / 4⌉

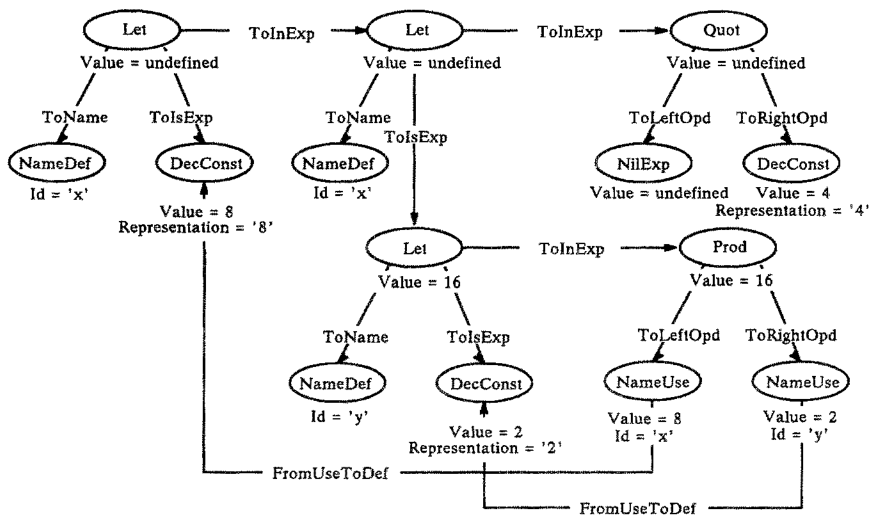


Figure 2 : Text and graph representation of an 'Exp' sentence.

- Map the contextfree syntactic structure of an 'Exp' sentence onto a tree-like graph structure which is equivalent to the sentence's abstract syntax tree. This structure builds the skeleton of the graph representation and it contains a separate **labeled node** for the root of any subexpression of the sentence. Distinguish different types of subexpressions by labeling their root nodes with the operators 'NilExp', 'Sum' etc. of the abstract syntax definition.

- Represent the abstract syntax tree's contextfree relationships, only implicitly defined within the abstract syntax, by **labeled edges** with arbitrarily chosen labels ('ToLeftOpd', ...) and introduce an additional type of edges for any kind of contextsensitive relationships. In our example we have to introduce edges labeled 'FromUseToDef' to bind applied occurrences of names to their value defining expressions.

- Use **(external) node attributes** to hold instances of phyla whose values are atomic from the current point of view and which encode properties inherent to a single node. This holds true for all phyla which do not appear on the left-hand side of any rule of the abstract syntax (phyla whose instances represent lexical units). Within our example we have to attach a 'String' attribute to all nodes labeled with one of the operators 'DecConst', 'NameDef', and 'NameUse'. To emphasize a distinction between strings representing identifiers and strings representing numbers, we introduce an attribute called 'Representation' for nodes labeled 'DecConst' and an attribute called 'Id' for nodes labeled either 'NameDef' or 'NameUse'.

- Use **(derived) node attributes** to encode node properties usually concerning aspects of dynamic semantics. Such an attribute is called 'derived' – instead of 'external' – if and only if its value is defined by a directed equation. Within this equation other attributes of the same node or of adjacent nodes may be referenced. Thus, we are able to establish functional attribute dependencies like: The 'Value' attribute of a 'Prod' node must be equal to the product of the 'Value' attributes of its two operand nodes which are the sinks of the two outgoing 'ToLeftOpd' and 'ToRightOpd' edges.

2.2 'Programming' with Graph Rewrite Rules

In the previous subsection I explained PROGRESS' underlying data model and discussed the matter of systematically deriving graph representations for the sentences of the language 'Exp'. This subsection deals with the subject of **specifying complex operations** mapping one sentence of the language 'Exp' onto another one of the same language. Thinking in terms of our data model these operations are nothing else but class preserving graph transformations.

So-called **subgraph tests** and **graph rewrite rules** are the basic building blocks for the definition of **complex graph transformations**. The former are boolean functions which test for the occurrence of a certain subgraph (pattern) within a host graph, and the latter are graph transformations which search for a certain subgraph within a host graph and replace this subgraph by another one. Usual control constructs known from imperative programming languages may be used to compose very complex graph transformations out of these tests and graph rewrite rules.

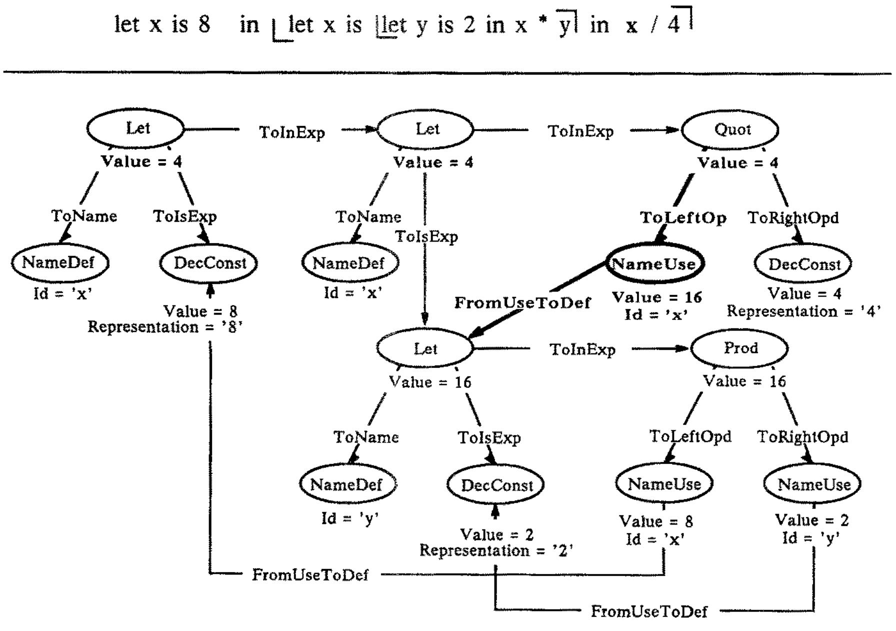


Figure 3 : Result of a rewrite rule's application to the graph of figure 2.

For a brief description of graph rewrite rules let us consider just one typical example. This is the rule 'replace an unexpanded subexpression by an applied occurrence of 'x' '. Figure 3 displays the effect of the application of such a rule onto the example displayed in figure 2. The execution of this rule (specified in section 3.2, figure 9) may be divided into the following five steps:

- **Subgraph test**: Select any subgraph within the host graph complying with the afore-mentioned requests, i.e. in this case any node labeled 'NullExp' within our tree-like graph structure. If there is none, return with failure.

- **Subgraph replacement**: Erase the selected subgraph including all incoming and outgoing edges, i.e. the node labeled 'NullExp' and one edge labeled 'ToLeftOpd', and insert the nodes and edges of the new subgraph, i.e. the node labeled 'NameUse'.
- **Embedding transformation**: Connect the new subgraph with the remainder of the host graph by incoming and outgoing edges, i.e. by two new edges labeled 'ToLeftOpd' and 'FromUseToDef'. So-called path expressions are used to determine the sometimes far-away located sources and targets of embedding edges within the host graph.
- **Attribute transfer**: Assign values to the external attributes of the nodes of the new subgraph, i.e. assign the value 'x' to the attribute 'Id' of the new node.
- **Attribute revaluation**: Compute the new values of all derived attributes within the new graph, i.e. the values of all 'Value' attributes, or at least of those derived attributes to which we have to assign new values.

3. The Specification Language PROGRESS

After this informal introduction to the basic principles of programmed graph rewriting systems, it's now time to focus our interest onto the **specification language PROGRESS** itself. For a first survey it should suffice to present only a typical subset of the language and to explain this subset by means of examples instead of formal definitions. The first subsection deals with the **declarative programming constructs** of PROGRESS which are used to specify graph schemes, whereas the second subsection deals with **graph queries and transformations**.

3.1 Defining Graph Schemes with PROGRESS

The scheme definition for a new class of graphs starts with the declaration of a set of attribute domains and a family of strict n-ary functions on these attribute domains (see figure 4). We shall skip the definition (implementation) of **attribute domains and functions** and assume a Modula-2 like host language for this purpose.

```
attribute_type Integer, String;
attribute_function undefined : Integer;
                   empty : String;
                   decToInt : ( String ) -> Integer;
                   mult : ( Integer, Integer ) -> Integer;
                   div : ( Integer, Integer ) -> Integer;
```

Figure 4: Declaration of attribute domains and functions.

Based on these (incomplete) attribute declarations, type declarations of three different categories characterize our abstract data type's graph representation. **Node and edge type declarations** are used to introduce type labels for nodes and edges, whereas **node class declarations** play about the same role as phyla, their counterparts within the operator/phylum based description of the language 'Exp'.

```
class EXP
  derived Value : Integer;
end;

class NAME
  external Id : String;
end;

class DEF_EXP is_a EXP end;

edge_type ToName : DEF_EXP -> NAME;

edge_type ToIsExp : DEF_EXP -> EXP;

edge_type ToInExp : DEF_EXP -> EXP;

class NAME_EXP is_a NAME, EXP end;

edge_type FromUseToDef : NAME_EXP -> EXP;

class BINARY_EXP is_a EXP end;

edge_type ToLeftOpd, ToRightOpd : BINARY_EXP -> EXP;

class CONST_EXP is_a EXP
  external Representation : String;
end;

class NIL_EXP is_a EXP end;

class MARKER end;

edge_type ToMarkedExp : MARKER -> EXP;
```

Figure 5: Declaration of node classes, edge types, and attributes.

Primarily, **node classes** are used to denote coercions of node types with common properties (following the lines of IDL /Ne 86/). Thereby, they introduce the concepts of classification and specialization into our language, and they eliminate the needs for duplicating declarations by supporting the concept of **multiple inheritance** along the edges of a class hierarchy. Additionally, node classes play the role of second order types. Being considered as types of node types, they support the controlled use of **formal (node) type parameters** within generic subgraph tests and graph rewrite rules (cf. figure 8 and 9). The class 'EXP' e.g. is the type of all those node types, whose nodes possess the derived 'Integer' attribute 'Value' (see figure 5). This holds true for all node types with the exception of 'NameDef' and 'Cursor'. The class 'BINARY_EXP', a subclass of the class 'EXP', is the type of all those node types, whose nodes may be sources of edges typed 'ToLeftOpd' and 'ToRightOpd'.

In the presence of the smallest class, the empty set, and the largest class, comprising all node types, this family of sets (the **class hierarchy**) has to form a **lattice** with respect to the ordering of sets by inclusion (corresponding to the 'is_a' relationship). This request enforces a disciplined use of the concept of multiple inheritance. Furthermore, it was a precondition for the development of a system of type compatibility constraints for PROGRESS.

```
node_type NameDef: NAME end;

node_type NameUse : NAME_EXP
  Value := [ that -FromUseToDef->.Value | undefined ];
end;

node_type Let : DEF_EXP
  Value := that -ToInExp->.Value;
end;

node_type Prod : BINARY_EXP
  Value := mult( that -ToLeftOpd->.Value, that -ToRightOpd->.Value);
end;

node_type DecConst : CONST_EXP
  Value := decToInt( .Representation);
end;

node_type NilExp : NIL_EXP
  Value := undefined;
end;

node_type Cursor : MARKER end;
```

Figure 6: Declaration of node types, and attribute dependencies.

The main purpose of a **node type declaration** is to define the behaviour of the nodes of this type, i.e. the set of all directed attribute equations, locally used for the (re-) computation of the node's derived attribute values. Let's start our explanations of the definition of **functional attribute dependencies** with the declaration of the node type 'DecConst' (in figure 6). Its 'Value' attribute is equal to the result of the function 'decToInt' applied to its own 'Representation' attribute. To compute the 'Value' of a 'Prod' node, we have to multiply the corresponding attributes of its left and right operand, i.e. the sinks of the outgoing edges typed 'ToLeftOpd' and 'ToRightOpd', respectively. In this case we assume that any 'Prod' node is the source of exactly one edge of both types and we strictly prohibit the existence of more than one outgoing edge of both types (indicated by the key word '**that**').

Due to the fact that we cannot always guarantee the existence of at least one edge, we had to introduce the possibility to define **sequences of alternative attribute expressions**. The general rule for the evaluation of such a sequence is as follows: try to evaluate the first alternative. If this fails due to the absence of an (optional) edge, then try to evaluate the next alternative. We request that at least the evaluation of the last alternative succeeds. Therefore, the 'Value' of a 'NameUse' node is either the 'Value' of its defining expression or, in the absence of a corresponding definition, 'undefined'.

3.2 Defining Graph Queries and Transformations with PROGRESS

Being familiar with PROGRESS' declarative style of programming, we are now prepared to deal with the operational constructs of the specification language. Some of these constructs form a partly textual, partly graphic **query sublanguage** for the definition of graph traversals which is very similar to data base query languages like /EW 83/. Elements of this sublanguage, in the sequel called path expressions, are mainly used to denote rather complex context conditions within subgraph tests and graph rewrite rules. Formal definitions of previous versions of this sublanguage have been published in /EL 87, Le 88a/.

```
path_op father : EXP -> EXP =
  <-ToLeftOpd- or <-ToRightOpd- or <-ToIsExp- or <-ToInExp-
end;

path_op nextValidDef : EXP -> DEF_EXP =
 (* Computes the root of the next valid surrounding definition. *)
    { not <-ToInExp- : father }
  & <-ToInExp-
end;

path_op binding ( Id : String ) : EXP -> EXP =
 (* Computes the root of the next visible surrounding definition of the name 'Id' *)
   nextValidDef
  & { not definition(Id) : nextValidDef }
  & -ToIsExp->
end;

path_op definition ( IdPar : String ) : DEF_EXP -> NAME  =  1 => 2  in
```

$$\begin{tikzcd}\boxed{\text{1: DEF\_EXP}} \arrow[r, "\text{ToName}"] & \boxed{\text{2: NAME}}\end{tikzcd}$$

```
                                      (* Applied to a set of 'DEF_EXP' nodes
                                         it computes the set of all those
                                         appertaining 'NAME' nodes whose
  condition 2.Id = IdPar ;               'Id' attributes are equal to 'IdPar'.  *)
end;
```

Figure 7: Definition of graph traversals, using textual and graphical path expressions.

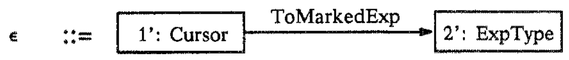
Path expressions may either be considered to be derived binary relationships or to be (node-) set-valued functions. The path expression '-ToLeftOpd->' e.g., used in figure 6 for the formulation of attribute dependencies, maps a set of nodes onto another node set; the elements of this second node set are sinks of 'ToLeftOpd' edges starting form nodes within the first node set. The expression '<-ToLeftOpd-' simply exchanges the roles of sinks and sources. Figure 7 contains the **textual definition** of three parameterized functional abstractions of path expressions, using conditional repetition ('{ <condition> : <path-expression> }'), composition ('&'), and union ('**or**'), and one example of a **graphic definition**. These path expressions specify the binding relationships between applied occurrences of identifiers ('NAME_EXP' nodes) and their corresponding value defining expressions ('EXP' nodes). The abstraction 'nextValidDef' e.g., applied to a set of nodes, computes the set of all those nodes that may be reached from elements of the first node set by: (1) following 'ToLeftOpd', 'ToRightOpd', and 'ToIsExp' edges form sinks to sources (cf. declaration of 'father'), as long as there is no incoming 'ToInExp' edge at any node on this path, (2) and finally following a 'ToInExp' edge from sink to source.

Skipping any further explanations concerning path expressions, we come to the specification (of a small subset) of the operations provided by the abstract data type 'ExpGraphs'. It is worthwhile to notice that node type parameters keep this part of the specification independent from the existence of any particular node type with the exception of the node type 'Cursor'. A unique node of this type is used to mark that subtree within a graph, called **current** subtree, that should be affected by the application of a graph rewrite rule. Thus, the declarations of figure 8 and 9 (together with the declarations of figure 5) may be used to specify a whole family of quite different desk calculators.

Let's start with the explanation of the operation 'Initialize' of figure 8. This operation creates the root of a tree-like subgraph if and only if the graph of interest contains no nodes of a type belonging to the class 'EXP'. We use the **test** 'ExpressionInGraph' to check this condition. The subsequent application of an instantiation of the graph rewrite rule (**production**) 'CreateExpressionRoot' never fails due to the fact that this rule has an empty matching condition. Being instantiated with any node type of the class 'NIL_EXP' its execution simply adds one node of this type, another node of the type 'Cursor', and an edge of the type 'ToMarkedExp' to a graph.

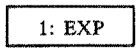
The production 'MoveCursorUp' is an example of a rewrite rule which **identifies** (all) **nodes** of its lefthand side with (all) nodes of its righthand side. An application of this rewrite rule neither deletes or modifies any node nor removes any not explicitly mentioned edge from the host graph. A successful application of this rule only redirects one edge of the type 'ToMarkedExp' so that its new sink is the father of its former sink, a condition that is expressed by the path operator 'father' within the rule's lefthand side.

```
production CreateExpressionRoot < ExpType : NIL_EXP > =
```



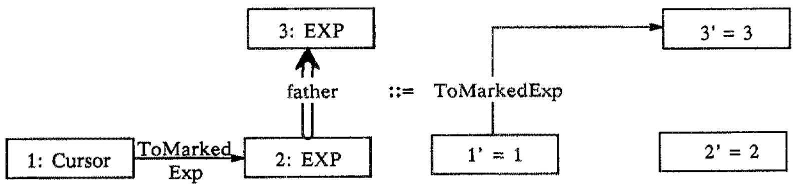
```
end ;

test ExpressionInGraph =
```



```
end ;
transaction Initialize < ExpType : NIL_EXP > =
  (* Creates the root of a new expression tree if there is no expression
     in the graph *)
  if not ExpressionInGraph then
     CreateExpressionRoot < ExpType >
  end (* if *)
end ;

production MoveCursorUp =
```

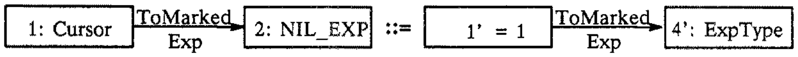


```
end ;
```

Figure 8: Declaration of simple productions, tests, and transactions.

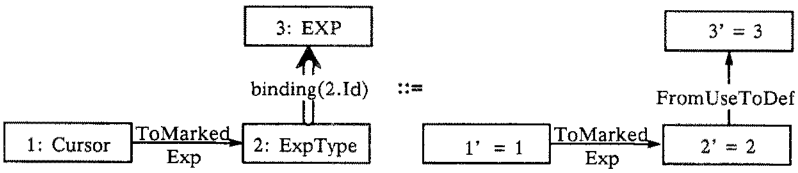
(Complex) graph transformations (like 'Initialize' of figure 8 or 'CreateAndBindNameExp' of figure 9) are termed **transactions** in order to indicate their **atomic** character. Similar to single graph rewrite rules whole transactions either cause consistency preserving graph transformations or abort without any modifications of the graph they were applied to. A transaction has to abort if one of the transactions or graph rewrite rules it's composed of abort. **Abortion** (**failure**) of a single graph rewrite rule occurs if the rule's lefthand side doesn't match with any subgraph of the graph it's applied to.

```
production CreateNameExp < ExpType : NAME_EXP > ( IdPar : String ) =
```



```
  embedding
     redirect <-ToLeftOpd-, <-ToRightOpd-, <-ToIsExp-, <-ToInExp-   from 2 to 4';
  transfer    4'.Id := IdPar   (* Initializes the external attribute of the new node *)
end;

production BindNameExp < ExpType : NAME_EXP > =
  (* Binds an applied occurrence to it's value defining expression which is determined by
     the path expression 'binding(2.Id)'                                                     *)
```



```
end;

transaction  CreateAndBindNameExp < ExpType : NAME_EXP > ( IdPar : String ) =
  (* Replaces a node of the class 'NIL_EXP' by a node of the class 'NAME_EXP'
     and tries to bind this new name to the expression of the corresponding declaration. *)
  CreateNameExp < ExpType > ( IdPar )
& try  BindNameExp < ExpType > ( IdPar )
  |  (* mark applied occurrence without declaration *)  . . .
  end (* try *)
end;
```

Figure 9: Specification of the graph transformation of section 2.

Thus the application of the transaction 'CreateAndBindNameExp' (of figure 9) either fails or consists of two graph rewrite steps. The first one causes the replacement of the current 'NIL_EXP' node by a new node of the class 'NAME_EXP'. The second one either binds the new applied occurrence to its value defining expression ('**try** BindNameExp ...') or - in the case of failure of 'BindNameExp' - marks the new applied occurrence as to be erroneous. The execution of the second graph rewrite rule requires the evaluation of the non-trivial path operator 'binding(2.Id)' which determines the target of the 'FromUseToDef' edge, whereas the execution of the first graph rewrite rule contains an embedding transformation and an attribute transfer, the latter being the assignment of the new name's string representation to the corresponding node's 'Id' attribute. Both rewrite steps even may **trigger the revaluation of many derived attributes**. The application, explained in subsection 2.2 (transforming the graph of figure 2 to that of figure 3), for instance initiated the (re-) evaluation of four 'Value' attributes.

The above mentioned **embedding transformation** within the graph rewrite rule 'Create-NameExp' consists of one '**redirect**' clause. The purpose of this clause is to redirect any incoming edge belonging to the expression's abstract syntax tree skeleton from the former 'NIL_EXP' node to the new 'NAME_EXP' node. In addition to '**redirect**' clauses, '**remove**' and '**copy**' clauses may be used to remove edges from identically replaced nodes or to duplicate edges and attach them to the replacing subgraph's nodes.

4. Related Work

The work presented within this paper is the first attempt to design a **strongly typed** language which is based on the concepts of **programmed graph rewriting systems** (like that of /Gö 88/) and supports a **declarative style of programming** for the description of object structures (like IDL /Ne 86/, or the languages surveyed in /HK 87/). The development of the language's type concept has been influenced by polymorphic programming languages like HOPE /BM 81/ which combine the flexibility of typeless languages with the reliability of strongly typed languages. By relying on the concept of a stratified type system with an infinite hierarchy of type universes (cf. /CZ 84/) we were able to incorporate **typed type parameters** into our language and to avoid the theoretical pitfalls of reflexive type systems with the 'Type is the type of all types including itself' assumption (cf. /MR 86/). The concept of **specialization** enables us to build complex class hierarchies. The idea to restrict this concept to the construction of **class lattices** has already been used within rewrite systems (cf. /CD 87/) and logic programming languages (cf. /AK 84/).

In order to facilitate a comparison of PROGRESS with those languages based on the more popular formalism of attribute (tree) grammars, I have used the well-known expression tree example for demonstration purposes. Therefore, its necessary to emphasize that PROGRESS is **not restricted to** the specification of abstract data types with **tree-like representations** but even more adequate in the case of graph-like representations without any dominant tree-like substructure. Thus, it is almost impossible within the framework of attribute (tree) grammars to specify the restriction that a class hierarchy has to form a lattice, but it is a straight forward task to write an appropriate lattice test with PROGRESS (/He 89/). Comparing the PROGRESS approach to model tree-like structures with that kind of modeling inherent to other attribute (tree) grammar approaches, we discover at least three principle differences:

- Derivation trees corresponding to subsequent applications of rewrite rules (productions) are neither used to represent object structures themselves nor to represent additional informations about these structures (like /Ka 85, Sc 87/). As a consequence, directed attribute equations are used to describe functional **attribute dependencies between adjacent nodes of a given graph**, and not to describe attribute flows along the edges of a derivation tree. Therefore, we believe (in contrast to /KG 89/) that the specification of attribute dependencies and the specification of graph rewrite rules should be kept separate from each other.

- A common disadvantage of our approach and the proposal in /KG 89/ is that **we are not able to identify directions like 'up' and 'down'** within arbitrary graphs. Therefore, we cannot adopt the classification of what we call 'derived' attributes into 'inherited' and 'synthesized' ones (cf. /Re 84/). Thus, the development of non-naive (incremental) attribute evaluation algorithms is much more difficult than for the case of attribute (tree) grammars. For basic work on this problem the reader is referred to /AC 87, Hu 87/.

- A third difference is also a direct consequence of the fact that we are not restricted to the world of trees: Our data model refrains the specificator from the somewhat artificially different treatment of relationships representing either the **contextfree or the contextsensitive syntax** (usually called static semantics) of a sentence of the language 'Exp'. Using labeled directed edges for the representation of both kinds of relationships, we are not forced to put contextsensitive informations into complex structured attributes (being a severe handicap for any incremental attribute evaluation algorithm), like /RT 84/, or to escape to another formalism, like /HT 86/.

5. Summary

This paper contains a first presentation of the new specification language **PROGRESS** (for writing **PRO**grammed Graph **RE**writing System Specifications), its underlying formalism, and a model oriented approach to the specification of abstract data types. This approach, called **graph grammar engineering**, is a model oriented one due to the fact that we use

- **attributed, node and edge labeled, directed graphs** to model complex object structures,
- and **programmed graph rewriting systems** to specify operations in terms of their effect on these graph models.

Having fixed the presented version of PROGRESS' **programming-in-the-small part** a few months ago, we are now starting to evaluate first experiences with the language and to develop the language's **programming-in-the-large part**, supporting the decomposition of large specifications into separate, reusable, and encapsulated subspecifications. Therefore, the presented version of PROGRESS may be classified as the programming-in-the-small kernel of a (very high level programming) language offering **declarative and procedural** elements for

- a **data flow oriented** style of programming (by means of directed attribute equations),
- an **object oriented** style of programming (by supporting multiple inheritance and dynamic binding of attribute designators to their value defining equations),
- a **rule based** style of programming (by using graph rewrite rules),
- and an **imperative** style of programming (within transactions).

Last but not least, we have started to design and implement a **programming environment for PROGRESS**, based on our previous experiences with the construction of the Integrated Programming Support ENvironment **IPSEN**. This environment will be built on top of the non-standard data base management system **GRAS** (cf. /LS 88/) and will comprise at least a syntax-directed editor (see /He 89/) and an interpreter. Thus, in days to come we might be able to specify the functional behaviour of PROGRESS' environment with PROGRESS itself and to analyze and execute the environment's specification by means of its own implementation.

Acknowledgments

I am indebted to M. Nagl and C. Lewerentz for many stimulating discussions about PROGRESS, to B. Westfechtel for critical and constructive comments on this manuscript, to all students, willingly producing large specifications with PROGRESS, and especially to U. Cordts, P. Heimann, P. Hormanns, R. Herbrecht, M. Lischewski, R. Spielmann, C. Weigmann, and A. Zündorf for designing and implementing first parts of the PROGRESS programming environment.

References

/AC 87/ B.Alpern, A.Carle, B.Rosen, P.Sweeney, K.Zadeck: *Incremental Evaluation of Attributed Graphs*, T. Report CS-87-29; Providence, Rhode Island: Brown University

/AK 84/ H.Ait-Kaci: *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*, Ph.D. Thesis; Philadelphia: University of Pennsylvania

/BM 81/ R.M.Burstall, D.B.MacQueen, D.T.Sannella: *HOPE - An Experimental Applicative Language*, Technical Report CSR-62-80; Edinburgh University

/CD 85/ R.J.Cunningham, A.J.J.Dick: Rewrite Systems on a Lattice of Types, in Acta Informatica 22, Berlin: Springer Verlag, pp. 149-169

/CZ 84/ R.Constable, D.Zlatin: *The Type Theory of PL/CV3*, in ACM TOPLAS, vol. 6, no. 1, pp. 94-117

/EL 87/ G.Engels, C.Lewerentz, W.Schäfer: *Graph Grammar Engineering - A Software Specification Method*, in Ehrig et al. (Eds.): Proc. 3rd Int. Workshop on *Graph Grammars and Their Application to Computer Science*, LNCS 153; Berlin: Springer Verlag, pp. 186-201

/En 86/ G.Engels: *Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung*, PH.D. Thesis; Düsseldorf: VDI-Verlag;

/ES 85/ G.Engels, W.Schäfer: *Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment*, in Ehrig et al. (Eds.): Proc. TAPSOFT '85, LNCS 186; Berlin: Springer-Verlag, pp. 179-193

/EW 83/ R.Elmasri, G.Wiederhold: *GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model*, in P.P.Chen (ed.): Entity-Relationship Approach to Information Modeling and Analysis, Amsterdam: Elsevier Science Publishers B.V. (North-Holland), pp. 49-72

/GJ 82/ H.J.Genrich, D.Janssens, G.Rozenberg, P.S.Thiagarajan: *Petri nets an their relation to graph grammars*, in Ehrig et al.: Proc. 2nd Int. Workshop on *Graph Grammars and Their Application to Computer Science*, LNCS 153; Berlin: Springer Verlag, pp. 115-142

/Gö 88/ H.Göttler: *Graphgrammatiken in der Softwaretechnik*, IFB 178; Berlin: Springer-Verlag;

/He 89/ R.Herbrecht: *Ein erweiterter Graphgrammatik-Editor*, Diploma Thesis; University of Technology Aachen;

/HT 86/ S.Horwitz, T.Teitelbaum: *Generating Editing Environments Based on Relations and Attributes*, in Proc. ACM TOPLAS, vol. 8, no. 4, pp. 577-608

/Hu 87/ S.E.Hudson: *Incremental Attribute Evaluation: An Algorithm for Lazy Evaluation in Graphs*, Technical Report TR 87-20; Tucson: University of Arizona

/HK 87/ R.Hull, R.King: *Semantic Database Modeling: Survey, Applications, and Research Issues*, in ACM Computing Surveys, vol. 19, No. 3, pp. 201-260

/JF 84/ G.F.Johnson, C.N.Fischer: *A Metalanguage and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors*, in Proc. ACM Symp. POPL '84

/Ka 85/ M.Kaul: *Präzedenz Graph-Grammatiken*, PH.D. Thesis; University of Passau

/KG 89/ S.M.Kaplan, St.K.Goering: *Priority Controlled Incremental Attribute Evaluation in Attributed Graph Grammars*, in Diaz, Orejas (Eds.): Proc. TAPSOFT '89, vol. 1, LNCS 351, Berlin: Springer Verlag, pp.306-320

/La 66/ P.J.Landin: *The next 700 programming languages*, Com. ACM 9, pp. 157-164

/Le 88/ C.Lewerentz: *Extended Programming in the Large in a Software Development Environment*, Proc 3rd ACM SIGPLAN/SIFSOFT Symp. on Practical Software Engineering Environments

/Le 88a/ C.Lewerentz: *Interaktives Entwerfen großer Programmsysteme*, PH.D. Thesis, IFB 194; Berlin: Springer-Verlag;

/LS 88/ C.Lewerentz, A.Schürr: *GRAS, a Management System for Graph-like Documents*, in C.Beeri et al. (Eds.): Proc. 3rd Int. Conf. on Data and Knowledge Bases; Los Altos, California: Morgan Kaufmann Publishers Inc., pp. 19-31

/Me 82/ A.Meier: *A Graph-Relational Approach to Geographic Databases*, in /ENR 82/, pp. 245-254

/MR 86/ A.R.Meyer, M.B.Reinhold: *'Type' is not a type*, Proc. 13th ACM Symp. POPL '86, pp. 287-295

/MW 84/ U.Möncke, B.Weisgerber, R.Wilhelm: *How to Implement a System for the Manipulation of Attributed Trees*, in U.Ammann (Ed.): Programmiersprachen und Programmentwicklung, IFB 77; Berlin: Springer Verlag

/Na 79/ M.Nagl: *Graph-Grammatiken: Theorie, Implementierung, Anwendungen*; Braunschweig: Vieweg-Verlag

/Na 85/ M.Nagl: *Graph Technology Applied to a Software Project*, in Rozenberg, Salomaa (Eds): *The Book of L*; Berlin: Springer-Verlag, pp. 303-322,

/Ne 86/ J.Newcomer: *IDL: Past Experience and New Ideas*, in Conradi et al. (Eds.): *Advanced Programming Environments*, LNCS 244; Berlin: Springer-Verlag, pp. 257-289

/No 87/ K.Normark: *Transformations and Abstract Presentations in Language Development Environment*, Technical Report DAIMI PB-222; Aarhus University

/Re 84/ T.Reps: *Generating Language-Based Environments*, PH.D. Thesis; Cambridge, Mass.: MIT Press

/RT 84/ T.Reps, T.Teilbaum: *The Synthesizer Generator*, in Proc. ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments, pp. 42-48

/Sc 87/ A.Schütte: *Spezifikation und Generierung von Übersetzern für Graph-Sprachen durch attributierte Graph-Grammatiken*, PH.D. Thesis; Berlin: EXpress-Edition

/So 84/: J.F.Sowa: *Conceptual Structures: Information Processing in Minds and Machines*; Reading, Mass.: Addison-Wesley

/TS 86/ G.Tinhofer, G.Schmidt (Eds.): Proc. WG '86 Workshop on *Graph-Theoretic Concepts in Computer Science*, LNCS 246; Berlin: Springer-Verlag