Aachener Informatik-Berichte AIB 94-13

Logic Based Programmed Structure Rewriting Systems

Andy Schürr
Lehrstuhl für Informatik III,
RWTH Aachen, Ahornstr. 55, D-52074 Aachen
e-mail: andy@i3.informatik.rwth-aachen.de

Abstract. This paper presents a new logic based framework for the formal treatment of graph rewriting systems as special cases of programmed rewriting systems for arbitrary relational data structures. Considering its expressive power, the new formalism surpasses almost all variants of nonparallel algebraic as well as algorithmic graph grammar approaches by offering set-oriented pattern matching facilities as well as nonmonotonic reasoning capabilities for checking pre- and postconditions of rewrite rules. Furthermore, the formalism closes the gap between the operation-oriented manipulation of data structures by means of rewrite rules and the declaration-oriented description of data structures by means of logic based languages. Finally, the formalism even offers recursively defined (sub-)programs, by means of which the application of rewrite rules may be regulated. A denotational semantics definition for these (sub-)programs relies on a special variant of fixpoint theory for noncontinuous but monotonic functions.

1. Introduction

Modern software systems for application areas like software engineering ore computer integrated manufacturing (CIM) are usually highly interactive and deal with complex structured objects. The systematic development of these systems requires precise and readable descriptions of their desired behavior. Many specification languages have been proposed to produce formal descriptions of various aspects of software systems, such as the design of object structures, the effect of operations on objects, or the synchronization of concurrently executing tasks. Many of these languages use *special classes of graphs* as their underlying data model and/or rule-oriented formalisms for the specification of complex transformation or inference processes on these graph-like data structures.

Within the software engineering research project IPSEN and the CIM research project SUKITS both graphs and rules in the form of *graph rewriting systems* are used for modeling the internal structure of abstract data types and for producing executable specifications of their interface operations [11, 14, 25]. The development of such a specification consists of two closely related subtasks. The first one is to design a graph model for the corresponding object structure. The second one is to program object (graph) analyzing and modifying operations by composing sequences of subgraph tests and graph rewrite rules.

Based on our experiences with graph rewriting systems, software specification languages, and programming environments, we started the design of a *graph rewriting system based specification language* and the development of an *integrated set of tools* for this language several years ago.

One result of this design process is the language PROGRES - an acronym for "PROgrammed Graph Rewriting System" - which possesses an almost static and polymorphic type system as well as a formally defined syntax and semantics. This language represents the first attempt to combine pure graph rewriting systems with new elements for the declaration of graph schemes and consistency constraints as well as for the definition of derived relations and attributes. As a consequence, we were forced to develop a *new framework* for the formal definition of our language which allows us to express

- derived graph properties and global consistency constraints,
- positive and negative pre- and postconditions for graph rewrite rules,
- deletion of nodes with unknown context,
- complex embedding transformation rules,
- and programming with parametrized graph rewrite rules.

This framework is based on *nonmontonic logic* and has some similarities to underlying theories of knowledge representation languages or deductive database systems [20, 30]. To summarize, the acronym PROGRES is a "trade mark" for an integrated set of tools, a very high level programming as well as specification language, and its underlying formalism. The main purpose of this paper is to present a generalized version of the underlying formalism [35], a new *structure rewriting approach*, whereas previously published papers had their main focus on the language PROGRES and its integrated programming environment [29, 34]. Note that due to size limits a couple of definitions are presented in simplified versions and the aspect of graph rewrite rule parameters had to be omitted (see [35] for more detailed informations).

In order to emphasize that PROGRES has mainly been developed for specifying and implementing software engineering tools, we will use the definition of a *syntax-directed editor* for the applicative language "Exp" as a running example within this paper. This language characterizes all legal input sequences for a very primitive desk calculator. The well-known "let"-operator is used to name and reuse intermediate computation results and the usual scoping rules for block structured programming languages direct the binding of applied occurrences of names to their corresponding definitions. Figure 1 defines the "Exp" language's syntax and figure 2 presents the text as well as the *graph representation* of a typical sentence of this language, which is composed of the sentence's abstract syntax tree skeleton and additional edges for binding applied identifier occurrences to their declarations. It contains one unexpanded subexpression with an undefined value represented by "(<Exp>)". As a consequence, the values of two subexpressions starting with "let x := ... " are undefined, too. And the computation of the subexpression "let y := 2 in x * y end" yields the value "16" due to the fact that "x" is bound to "8" (by the outermost "let") and "y" to "2".

(1)	EXP ::=	IntLiteral | DefExp | ApplId | PlusExp | MinusExp | MultExp | DivExp ;
(2)	IntLiteral ::=	Text ; *(* Text is a sequence of digits *)*
(3)	DefExp ::=	"let" DeclId ":=" EXP "in" EXP "end" ;
(4)	DeclId ::=	Name ; *(* Name is a sequence of characters *)*
(5)	ApplId ::=	Name ;
(6)	PlusExp ::=	EXP "+" EXP ;
(7)	. . .	

Fig. 1: Syntax of the language "Exp"

The rest of this paper is organized as follows:

- Section 2 repeats some basic terminology of predicate logic and contains the formal definition of structures as sets of formulas, with graphs being special cases of structures.
- Section 3 presents the definition of structure schemes as sets of formulas, too.
- Section 4 is dedicated to the formal definition of substructures and redex selection (a redex is that part of a structure which matches a rewrite rule's left-hand side).
- Section 5 introduces the new structure rewriting formalism itself.
- Section 6 deals with recursively defined structure rewriting programs, so-called transactions, and their denotational semantics definition.
- And the last section of the paper reviews the main motivation for the development of yet another structure rewriting approach and lists a number of open problems.

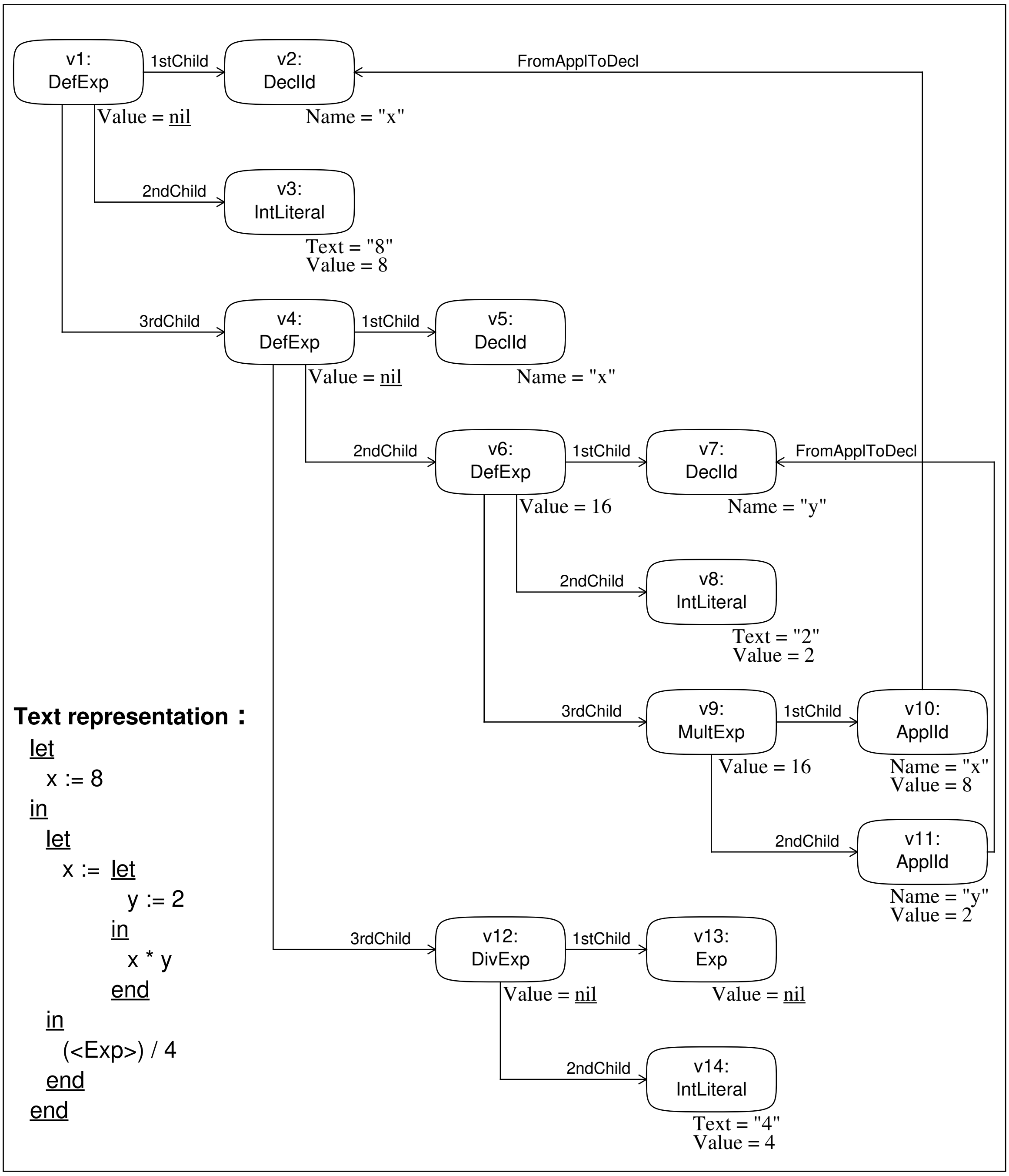


Fig. 2: Text and graph representation of an "Exp" sentence

2. Predicate Logic Formulas, Structures, and Graphs

This section introduces basic terminology of predicate logic and explains the *modelling of attributed node and edge labeled graphs* as special cases of structures, i.e. as sets of atomic formulas. Note that all definitions and propositions of the following sections are independent of the selected graph model and its encoding. Therefore, our structure rewriting approach is able to deal with about the same class of relational structures as [22], may easily be instantiated with a different graph model as e.g. the hypergraphs of [9], and should even be adaptable to the case of almost arbitrary knowledge bases as e.g. defined in [20, 30].

Definition 2.1 Signature.

A 6-tuple $\Sigma := (\mathcal{L}_F, \mathcal{L}_P, \mathcal{L}_C, \mathcal{V}, \mathcal{W}, \mathcal{X})$ is a *signature* iff:

(1) $\mathcal{L}_F$ is an alphabet of function symbols[1].
(2) $\mathcal{L}_P$ is an alphabet of predicate symbols[2].
(3) $\mathcal{L}_C$ is an alphabet of constant symbols (for attribute values, type identifiers etc.).
(4) $\mathcal{V}$ is a special alphabet of object (node) identifiers.
(5) $\mathcal{W}$ is a special alphabet of identifiers for sets of objects (nodes).
(6) $\mathcal{X}$ is an alphabet of logical variables used for quantification purposes. ■

Example 2.2 Signature for an "Exp"-Graph.

The graph signature for fig. 2 is $\Sigma := (\mathcal{L}_F, \mathcal{L}_P, \mathcal{L}_C, \mathcal{V}, \mathcal{W}, \mathcal{X})$ with:

(1) $\mathcal{L}_F$:= { +, -, … } .
(2) $\mathcal{L}_P$:= { <, … , *rel*, *type*} .
(3) $\mathcal{L}_C$:= { 0, 1, … , integer, DefExp, … , Value, FromApplToDecl, … } .
(4) $\mathcal{V}$:= { v1, v2, … } .
(5) $\mathcal{W}$:= { w1, w2, … } .
(6) $\mathcal{X}$:= { x1, x2, … } . ■

Note that the alphabet $\mathcal{L}_P$ contains two predicate symbols which are very important for the more or less arbitrarily selected encoding of graphs:

- The predicate symbol "*type*" with: "*type*(x, vl)" for "node x has type/label vl" or with"*type*(τ, t)" for "term τ has type t", and
- The predicate symbol "*rel*" with: "*rel*(x, el, y)" for "edge with type/label el connects source node x to target node y" or with "*rel*(x, a, τ)" for "attribute a at node x has value τ".

In the sequel we will always assume that Σ is a signature over the above mentioned alphabets.

Definition 2.3 Σ-Term.

TERM(Σ) is the set of all terms (in the usual sense) which contain function symbols of $\mathcal{L}_F$, free variables of $\mathcal{X}$, and constant symbols of $\mathcal{L}_C$, $\mathcal{V}$, and $\mathcal{W}$. ■

Definition 2.4 Σ-Atom.

ATOM(Σ) is the set of all atomic formulas over TERM(Σ) which contain predicate symbols of $\mathcal{L}_P$ and additionally "=" for expressing the equality of two Σ-terms. ■

---

1) More precisely, a family of alphabets of symbols if we take domain and range of functions into account.
2) A family of alphabets of symbols if we take arities of predicates into account.

Definition 2.5 Σ-Formula.

FORM(Σ) is the set of all first order predicate logic formulas with ATOM(Σ) as atomic formulas, ∧, ∨, … as logical connectives, and ∃, ∀ as quantifiers. ■

Definition 2.6 Derivation of Formulas.

With Φ and Φ' being sets of Σ-formulas

$$\Phi \vdash \Phi'$$

means that all formulas of Φ' are *derivable* from the set of formulas Φ using any (consistent, complete) inference system of first order predicate logic with equality. ■

Definition 2.7 Closed Σ-Formulas.

The set $F \subseteq FORM(\Sigma)$ is a *closed set of* **Σ**-*formulas* (write: $F \in \mathcal{F}(\Sigma)$) ⇔
F contains only bound variables (of $\mathcal{X}$ in Σ)[3]. ■

In the following, elements of $\mathcal{F}(\Sigma)$ will be used to represent structures, structure schemes, scheme consistent structures, and even left- and right-hand sides as well as pre- and postconditions of structure rewrite rules.

Definition 2.8 Σ-Structure.

A closed set of formulas $G \in \mathcal{F}(\Sigma)$ is a **Σ**-*structure* (write: $G \in \mathcal{G}(\Sigma)$) ⇔
$G \subseteq ATOM(\Sigma)$ and G does not contain formulas of the form "$\tau_1 = \tau_2$". ■

Example 2.9 An "Exp"-Graph-Structure.

The following structure is a set of formulas which has the graph G of fig. 2 as a model but which also has many other graphs as models, in which we are not interested in:

G := { *type*(v1, DefExp), *type*(v2, DeclId), *type*(v3, IntLiteral), … ,
rel(v1, 1stChild, v2), *rel*(v1, 2ndChild, v3), … , *rel*(v3, Value, 8), … } . ■

The next section introduces a so-called "completing operator" which allows us to get rid of unwanted graph models and to reason about properties of "minimal" (graph) models on a pure syntactical level. Therefore, models of structures are not introduced formally and will not be used within the rest of this paper.

3. Structure Schemes and Scheme Consistent Structures

This section is mainly dedicated to the definition of *structure schemes* - the term "structure scheme" is used in the same sense as the term "database scheme" - and scheme consistent structures. Such a scheme is a set of properties of a certain class of structures. Thus, PROGRES and its underlying theory support the separation of implicitly given integrity constraints, functional attribute dependencies etc. from graph (structure) rewrite rules. For this purpose, an extension of pure 1st-order predicate logic is necessary with respect to the capability of nonmonotonic reasoning:

3) Requiring bindings for all logical variables, we avoid any difficulties with different interpretations of free variables in different inference systems.

Example 3.1 Nonmonotonic Reasoning.

The singleton set G := { *type*(v1, ApplId) } is a structure which has all graphs containing at least one "ApplId" node as models. Being interested in properties of "minimal" G graph models only, we would like to be able to prove

$$\forall \text{ x, t: } \mathit{type}(\text{x, t}) \rightarrow (\text{ x = v1} \wedge \text{t = ApplId }),$$

i.e. that all models of G, we are interested in, contain exactly one node. One obvious solution for this problem is to add the above mentioned formula to our structure G. But our structure rewriting formalism is not able to rewrite sets of nonatomic formulas[4] and, therefore, structures are restricted to contain atomic formulas only (see definition 2.8). Therefore, we need a mechanism which allows us to use additional formulas without explicitly including them into our structures.

A related problem has been extensively studied within the field of deductive database systems and has been attacked by a number of quite different approaches either based on the so-called "*closed world assumption*" or using "*nonmonotonic reasoning*" capabilities (cf. [21, 26, 27]). The main idea of (almost) all of these approaches is to distinguish between basic facts and derived facts and to add only negations of basic facts to a rule base which are not derivable from the original set of facts.

It is beyond the scope of this paper to explain nonmonotonic reasoning in more detail. Therefore, we will simply assume the existence of a "*completing operator*" $\mathcal{C}$ which adds a certain set of additional formulas to a structure such that the resulting set of formulas is consistent and "sufficiently complete" to allow us to prove the above mentioned properties by using the axioms of "pure" first-order predicate logic only. For the example mentioned above the operator $\mathcal{C}$ might have the following form:

$$\begin{aligned}\mathcal{C}(\text{G}) := \text{G } & \cup \{ \neg \exists \text{ y: } \mathit{rel}(\text{v, r, y}) \mid \text{v} \in \mathcal{V}, \text{r} \in \mathcal{L}_\text{C}, \text{and for all v': } \mathit{rel}(\text{v, r, v'}) \notin \text{G} \} \\ & \cup \ldots \cup^5 \\ & \{ \forall \text{ x, l: } \mathit{type}(\text{x, l}) \rightarrow (\text{x} = \text{v}_1 \vee \ldots \vee \text{x} = \text{v}_\text{k}) \mid \text{v}_1, \ldots, \text{v}_\text{k} \in \mathcal{V} \text{are all id in G} \}.\end{aligned}$$

By means of this operator we are able to prove for instance that G contains only one node without any outgoing edges:

$$\mathcal{C}(\text{G}) \vdash (\forall \text{ x, l: } \mathit{type}(\text{x, l}) \rightarrow \text{x = v1 }) \wedge (\neg \exists \text{ y: } \mathit{rel}(\text{v1, FromApplToDecl, y})). \blacksquare$$

Definition 3.2 Σ-Structure Completing Operator.

A function $\mathcal{C}$: $\mathcal{G}(\Sigma) \rightarrow \mathcal{F}(\Sigma)$ is a Σ-*structure completing operator* ⇔

For all structures G ∈ $\mathcal{G}(\Sigma)$: G ⊆ $\mathcal{C}$(G) and $\mathcal{C}$(G) is a consistent set of formulas. ■

Definition 3.3 Σ-Structure Scheme.

A tuple S := (Φ, $\mathcal{C}$) is a Σ-*structure scheme* (write: S ∈ $\mathcal{S}(\Sigma)$) ⇔

(1) Φ ∈ $\mathcal{F}(\Sigma)$ is a consistent set of formulas without object (set) identifiers, i.e. it contains no propositions about individual objects.

(2) $\mathcal{C}$ is a Σ-structure completing operator. ■

---

4) The problem is to find a reasonable semantics for the deletion of nonatomic formulas which is compatible with the derivation process of a first order predicate logic inference system.

5) We have omitted the definition of many additional sets of formulas which for instance force different object identifiers to represent different objects (in order to exclude final graph models which map all constants onto the same value/object; for further details see [35]).

Definition 3.4 Scheme Consistent Structure.

Let S := (Φ, $\mathcal{C}$) ∈ $\mathcal{S}$(Σ) a scheme. A Σ-structure G ∈ $\mathcal{G}$(Σ) is *scheme consistent with respect to S* (write: G ∈ $\mathcal{G}$(S)) ⇔

$\mathcal{C}$(G) ∪ Φ is a consistent set of formulas. ■

Fig. 3 contains a cutout of the PROGRES definition of our "Exp" example's graph scheme with three different kinds of type declarations. *Node and edge type* declarations are used to introduce type labels for nodes and edges, whereas *node class* declarations form an inheritance hierarchy and represent types of node types with common properties. The class BINARY_EXP for instance is a subclass of the classes EXP and BINARY_NODE and inherits all properties from both superclasses. Furthermore, nodes of a type of this class are sources of 1stChild and 2ndChild edges and have a Value attribute (for further details see [34, 35]). The node type MultExp for instance inherits all properties from the class BINARY_EXP and additionally defines the directed equation for their Value attribute. This attribute is called a *derived attribute* in contrast to an *intrinsic attribute*, which has a value of its own and receives new values by means of assignments within rewrite rules only. Its value is derived from values of other attributes and will be kept consistent with these attribute values by means of an incrementally working lazy attribute evaluation process [17].

Finally, *path declarations* define derived relations by means of graph traversing expressions. The path 1stOpd for instance defines a relation between all BINARY_EXP nodes and EXP nodes which are connected by 1stChild edges. And the path visibleDecl, needed in figure 5, searches the declaration of a given identifier name by ascending in an expression's syntax tree from scope to scope as long as these nested scopes do not contain a declaration of the given name (the paths superScope and decl are omitted in figure 3 for lack of space only). Note that the operator "&" concatenates path expressions and curly braces "{ . . . }" denote repetitions of an expression following "?" as long as the condition before "?" is valid.

```
node class AST_NODE end; (* AST = Abstract Syntax Tree *)
node class BINARY_NODE is a AST_NODE end; (* AST_NODE with two children. *)
edge type 1stChild, 2ndChild: BINARY_NODE --> AST_NODE;
node class EXP is a AST_NODE;
   derived Value: integer;
end; (* The directed equations for the derived Value attribute are part of the type decls. below. *)
node class BINARY_EXP is a EXP, BINARY_NODE end;
path 1stOpd: BINARY_EXP --> EXP =  -1stChild-> & instance of EXP  end;
   (* Follows 1stChild edge and verifies that target belongs to subclass EXP of AST_NODE. *)
   . . .
path visibleDecl(OfName: string): AST_NODE --> DECL_ID =
      superScope & { not with decl(OfName) ? superScope } & decl(OfName)
end; (* Returns visible declaration of identifier name OfName (if existent). *)
node type MultExp: BINARY_EXP;
   redef derived Value = self.1stOpd.Value * self.2ndOpd.Value;
end; (* Value of MultExp = Value of its first Operand * Value of its second operand. *)
```

Fig. 3: The "Exp" graph scheme (cutout)

Example 3.5 A Graph Scheme for "Exp" graphs.

A graph (structure) scheme S := (Φ, $\mathcal{C}$) for the graph G in fig. 2, which corresponds to the declarations of fig. 3, has about the following form:

$$
\begin{array}{lll}
\Phi := \{ & \forall\, t{:}\ \mathit{type}(t, \mathsf{AST\_NODE})\ , & \\
& \forall t{:}\ \mathit{type}(t, \mathsf{BINARY\_NODE}) \rightarrow & \mathit{type}(t, \mathsf{AST\_NODE})\ , \\
& \forall t{:}\ \mathit{type}(t, \mathsf{BINARY\_EXP}) \rightarrow & \mathit{type}(t, \mathsf{EXP}) \wedge \mathit{type}(t, \mathsf{BINARY\_NODE})\ , \\
& \forall\, x, y{:}\ \mathit{rel}(x, \mathsf{1stChild}, y) \rightarrow & \exists\, t{:}\ \mathit{type}(x, t) \wedge \mathit{type}(t, \mathsf{BINARY\_NODE}) \\
& & \wedge\ \exists\, t{:}\ \mathit{type}(y, t) \wedge \mathit{type}(t, \mathsf{AST\_NODE}), \\
& \ldots\ , & \\
& \multicolumn{2}{l}{\forall\, x, y{:}\ \mathsf{1stOpd}(x, y) \leftrightarrow \mathit{rel}(x, \mathsf{1stChild}, y) \wedge \exists\, t{:}\ \mathit{type}(y, t) \wedge \mathit{type}(t, \mathsf{EXP})} \\
& \multicolumn{2}{l}{\forall\, v, x, y{:}\ \mathsf{visibleDecl}(v, x, y) \leftrightarrow \exists\, z, z'{:}\ \ \mathsf{superScope}(x, z)} \\
& & \qquad\qquad \wedge\ \mathsf{loop}(v, z, z') \wedge \mathsf{decl}(v, z', y), \\
& \forall\, v, x, y{:}\ \mathsf{loop}(v, x, y) \leftrightarrow & \exists\, z{:}\ \mathsf{decl}(v, x, z) \wedge x = y \\
& & \vee\ \neg\, \exists\, z{:}\ \mathsf{decl}(v, x, z) \\
& & \quad \wedge\ \exists\, z'{:}\ \mathsf{superScope}(x, z') \wedge \mathsf{loop}(v, z', y) \\
& \ldots\ , & \\
& \mathit{type}(\mathsf{MultExp}, \mathsf{BINARY\_EXP})\ , & \\
& \forall\, x, v{:}\ \ \mathit{type}(x, \mathsf{MultExp}) \wedge \mathit{rel}(x, \mathsf{Value}, v) & \\
& \quad \rightarrow \exists\, y1, v1, y2, v2{:}\ & \mathsf{1stOpd}(x, y1) \wedge \mathit{rel}(y1, \mathsf{Value}, v1) \\
& & \wedge\ \ \mathsf{2ndOpd}(x, y2) \wedge \mathit{rel}(y2, \mathsf{Value}, v2) \\
& & \wedge\ \ v = v1 * v2\ \}\ .
\end{array}
$$

$\mathcal{C}(G) := G \cup \ldots$. ■

4. Redex Selection with Additional Constraints

The main purpose of this section is to formalize the term "*redex under additional constraints*" by means of morphisms. This term is central to the definition of "structure rewriting with pre- and postconditions" of the next section and determines which substructures of a structure, called redices, are legal "targets" for a rewriting step, i.e. match a given rule's left-hand side.

Note that the embedding rules of PROGRES[6] are able to delete, copy, and redirect arbitrary large bundles of edges. We will handle these embedding rules by introducing special *object set identifiers* on a structure rewrite rule's left- and right-hand side which are allowed to match an arbitrarily large (maximal) set of object identifiers in a given redex (see definition 5.1 and example 5.2).

Therefore, morphisms between structures, which select the affected substructure/redex for a rewrite rule in a structure, are neither total nor partial functions. In the general case, these *morphisms are relations* between object (set) identifiers. Furthermore, these relations are required to preserve the properties of the source structure (e.g a rule's left-hand side) while embedding it into the target structure (e.g the structure we have to rewrite) as usual and to take additional constraints for the chosen embedding into account.

6) PROGRES embedding rules are slightly simplified variants of those belonging to the expression oriented algorithmic graph grammar approach à la Nagl [23].

Definition 4.1 Redex Selecting Relation (simplified).

With F, F' $\in \mathcal{F}(\Sigma)$ and $\mathcal{V}_F$, $\mathcal{W}_F$ and $\mathcal{V}_{F'}$, $\mathcal{W}_{F'}$ being those sets of object (set) identifiers which are used in F and F', respectively, a relation

$$u \subseteq (\mathcal{V}_F \times \mathcal{V}_{F'}) \cup (\mathcal{W}_F \times \mathcal{W}_{F'}) \cup (\mathcal{W}_F \times \mathcal{V}_{F'})$$

is a *redex selecting relation* (also called Σ-*relation*) from F to F' $\Leftrightarrow$

(1) For all $v \in \mathcal{V}_F$: $|u(v)| = 1$ (with $u(x) := \{ y \mid (x, y) \in u \}$),
i.e. every source object identifier will be mapped onto one target object identifier.

(2) For all $w \in \mathcal{W}_F$: $u(w) \subseteq \mathcal{W}_{F'}$ and $|u(w)| = 1$ or $u(w) \subseteq \mathcal{V}_{F'}$,
i.e. every source object set identifier will be mapped either onto one target object set identifier or onto a set of target object identifiers.

Let u* be the natural extension[7] of u to the domains of Σ-terms and Σ-formulas; furthermore, with $\Phi \subseteq \mathcal{F}(\Sigma)$, the following short-hand will be used in the sequel:

$$u^*(\Phi) := \{ \phi' \in \mathcal{F}(\Sigma) \mid \phi \in \Phi \text{ and } (\phi, \phi') \in u^* \} . \blacksquare$$

Example 4.2 A Redex Selecting Relation.

Let F, F' $\in \mathcal{F}(\Sigma)$ with $\mathcal{V}_F := \{ v1 \}$, $\mathcal{V}_{F'} := \{ v11, v12, v21, v22 \}$, $\mathcal{W}_F := \{ w1, w2 \}$, and with $\mathcal{W}_{F'} := \{\}$. Then $u := \{ (v1, v11), (w2, v21), (w2, v22) \}$ is a redex selecting relation from F to F': $u^*(\{ \mathit{type}(v1, t) \}) = \{ \mathit{type}(v11, t) \}$.
$u^*(\{ \mathit{rel}(v1, r, w1) \}) = \{\}$.
$u^*(\{ \mathit{type}(w2, t) \}) = \{ \mathit{type}(v21, t), \mathit{type}(v22, t) \}$. ■

Note that in the general case (defined in [35] as so-called "renaming" relations), redex selecting relations are allowed to map identifiers onto (sets of) Σ-terms. In this way, we are also able to parametrize structure rewrite rules with terms representing values for object properties (e.g. attributes of nodes) and to modify properties of already existing objects.

Definition 4.3 Σ-(Structure-)Morphism.

Let F, F' $\in \mathcal{F}(\Sigma)$. A Σ-relation u from F to F' is a Σ-*morphism* from F to F'
(write: $u: F \tilde{\Rightarrow} F'$) $\Leftrightarrow$

$$F' \models u^*(F) .$$

With F, F' $\in \mathcal{G}(\Sigma) \subseteq \mathcal{F}(\Sigma)$ being Σ-structures, u will be called Σ-*structure morphism*. ■

Proposition 4.4 The Category of Σ-Structures.

Assume "∘" to be the usual composition of binary relations. Then, $\mathcal{F}(\Sigma)$ together with the family of Σ-morphisms defined above and "∘" is a *category*; the same holds true for the set of closed atomic formulas $\mathcal{G}(\Sigma)$ and the family of Σ-structure morphisms.

Proof:

Σ–Morphisms are closed w.r.t. "∘", i.e. $u: F \tilde{\Rightarrow} F'$, $u': F' \tilde{\Rightarrow} F'' \Rightarrow (u \circ u'): F \tilde{\Rightarrow} F''$:

u and u' are Σ-relations	$\Rightarrow_{\text{def. 4.1}}$	$(u \circ u')$ is a Σ-relation
	and	$(u \circ u')^* = u^* \circ u'^*$.
u, u' are morphisms	$\Rightarrow_{\text{def. 4.3}}$	$F' \models u^*(F)$
	and	$F'' \models u'^*(F')$

7) u* relates two terms or formulas to each other, if both are identical with the exception of u-related object (set) identifiers.

$F' \vdash u^*(F) \quad \Rightarrow_{\text{renaming preserves proofs}} \quad u'^*(F') \vdash u'^*(u^*(F)) = (u \circ u')^*(F)$

$\qquad\qquad\qquad \Rightarrow_{\text{modus ponens}} \quad F'' \vdash (u \circ u')^*(F)$,

i.e. $(u \circ u')$ is a morphism from F to F".

Existence of neutral Σ–morphism id_F for any $F \in \mathcal{F}(\Sigma)$:

Obviously, the relation id_F, which maps any object (set) identifier in F onto itself, is a neutral element for the family of Σ-relations. Then,

$$id_F^*(F) = F \;\Rightarrow\; F \vdash id_F^*(F) \;\Rightarrow\; id_F: F \tilde{\rightarrow} F ,$$

i.e. id_F is the required neutral morphism.

Associativity of "∘" for Σ-morphisms:

Follows directly from the fact that "∘" is associative for binary relations.

In order to obtain the proof that the family of Σ-structure morphisms together with $\mathcal{G}(\Sigma)$ and "∘" is a category we simply have to replace any $\mathcal{F}(\Sigma)$ above by $\mathcal{G}(\Sigma)$. ■

Definition 4.5 Substructure.

$G, G' \in \mathcal{G}(\Sigma)$ are structures. G is a *substructure* of G' with respect to a Σ-relation u (write: $G \subseteq_u G'$) $\Leftrightarrow$ $\quad u: G \tilde{\rightarrow} G'$. ■

This definition coincides with the usual meaning of substructure (or subgraph), if G and G' do not contain object set identifiers (and u represents a bijective mapping).

Proposition 4.6 Soundness of Substructure Property.

For $G, G' \in \mathcal{G}(\Sigma)$ being structures, the following properties are equivalent:

G is a substructure of G' with respect to a Σ-relation u

$\Leftrightarrow$ $u^*(G)$ is a subset of G'.

Proof:

$G \subseteq_u G' \;\Leftrightarrow_{\text{see def. 4.5}}\; u: G \tilde{\rightarrow} G' \;\Leftrightarrow_{\text{see def. 4.3}}\; G' \vdash u^*(G) \;\Leftrightarrow_{u^*(G), G' \in \mathcal{G}(\Sigma)}\; u^*(G) \subseteq G'$.

The last step of the proof follows from the fact that G and G' are sets of atomic formulas without "=", such that "⊆" (normal set inclusion) and "⊢" are equivalent relations. ■

Definition 4.7 Substructure with Additional Constraints.

$S := (\Phi, \mathcal{C})$ is a Σ-structure scheme, $G, G' \in \mathcal{G}(S)$, and $F \in \mathcal{F}(\Sigma)$ is a set of constraints with references to object (set) identifiers of G only. G is a *constrained substructure* of G' with respect to a Σ-relation u and the additional set of constraints F (write: $G \subseteq_{u,F} G'$) $\Leftrightarrow$

(1) $G \subseteq_u G'$, i.e. $G' \vdash u^*(G)$, i.e. G is a substructure of G'.

(2) $u: F \cup G \tilde{\rightarrow} \Phi \cup \mathcal{C}(G')$, i.e. $\Phi \cup \mathcal{C}(G') \vdash u^*(F \cup G)$,

i.e. we are able to prove all constraints in F for the embedding of G in G' by using the basic facts of G' and the set of formulas generated by the completing operator $\mathcal{C}$ as well as the set of formulas Φ of the structure scheme S.

These conditions are equivalent to the existence of the diagram in figure 4, with inclusions i_1 and i_2 being the following morphisms:

$i_1: G \tilde{\rightarrow} F \cup G$, $\qquad$ because $i_1^*(G) = G \subseteq F \cup G$, i.e. $F \cup G \vdash i_1^*(G)$.

$i_2: G' \tilde{\rightarrow} \Phi \cup \mathcal{C}(G')$, $\quad$ because $i_2^*(G') = G' \subseteq \mathcal{C}(G')$, i.e. $\Phi \cup \mathcal{C}(G') \vdash i_2^*(G')$. ■

$$\begin{tikzcd} F \cup G \arrow[d, "u"'] & G \arrow[l, "i_1"'] \arrow[d, "u"] \\ \mathcal{C}(G') \cup \Phi & G' \arrow[l, "i_2"'] \end{tikzcd}$$

Fig. 4: Substructure (redex) selection with additional constraints

Example 4.8 Substructure Selection.

Let S := (Φ, $\mathcal{C}$) be an extended version of the scheme belonging to figure 3 which additionally contains the definition of a new predicate "unused":

Φ := { $\forall$y: unused(y) $\leftrightarrow \neg \exists$ x: *rel*(x, FromApplToDecl, y) , … } and

$\mathcal{C}$(G) := G $\cup$ { $\neg \exists$ x: *rel*(x, r, v') | not exists v: *rel*(v, r, v') $\in$ G } $\cup$ … .

Furthermore, G is a S-scheme consistent structure which represents a graph with two nodes and one edge:

G := { *type*('1, DefExp), *type*('2, DeclId), *rel*('1, 1stChild, '2) } .

And F is an additional embedding constraint for G:

F := { unused('2) } .

Then G is a substructure of G' of figure 2 with respect to F and the redex selecting relation

u := { ('1, v4), ('2, v5) }.

Proof:

With G' := { *type*(v4, DefExp), *type*(v5, DeclId), *rel*(v4, 1stChild, v5), … }

we are able to prove that

$\mathcal{C}$(G') $\cup \Phi \vdash$ u*(G) and $\neg \exists$ x: *rel*(x, FromApplToDecl, v5) $\in \mathcal{C}$(G') .

$\Rightarrow$ $\mathcal{C}$(G') $\cup \Phi \vdash$ { unused (v5) } . ■

5. Scheme Preserving Structure Rewrite Rules

Having presented the definitions of structure schemes, scheme consistent structures, and structure morphisms, we are now prepared to introduce *structure rewrite rules* as quadruples of *sets of closed formulas* and the application of structure rewrite rules as the construction of commuting diagrams in a similar way as it is done in the algebraic graph grammar approach. But note that due to the usage of redex selecting *relations* instead of (partial) mappings, and the necessarily slightly asymmetric definition of structure rewriting (object set identifiers in a rule's left-hand side must be mapped onto a maximal set of object identifiers in a structure) the most important properties of the algebraic approach get lost:

- Our (sub-)diagrams are not pushouts in the general case.
- The application of a structure rewrite rule is not invertible in the general case.

Unfortunately, we had to pay this price for the ability to formalize complex embedding rules within the language PROGRES, which delete/copy/redirect arbitrarily large edge bundles.

Definition 5.1 Structure Rewrite Rule.

A quadruple p := (AL, L, R, AR) with AL, AR $\in \mathcal{F}(\Sigma)$ and with L, R $\in \mathcal{G}(\Sigma)$ is a *structure rewrite rule (production)* for the signature Σ (write: p $\in \mathcal{P}(\Sigma)$) $\Leftrightarrow$

(1) The set of left-hand side application/embedding conditions AL contains only object (set) identifiers of the left-hand side L.

(2) The set of right-hand side application/embeddding conditions AR contains only object (set) identifiers of the right-hand side R.

(3) Every set identifier of L is also a set identifier in R (they are only used for deleting dangling references and for establishing connections between new substructures created by a rule's right-hand side and the remaining rest of the modified structure). ■

Consider the production ExpandToXIdent of figure 5, which contains almost all important components of PROGRES productions. This production replaces an unexpanded subexpression by an applied occurrence of the identifier "x" and binds the new identifier to the nearest enclosing declaration of "x". Its execution may be divided into seven steps:

(1) *Subgraph selection*: Find any subgraph/redex within the host graph consisting of two nodes with labels DeclId and Exp, respectively.

(2) *Check preconditions*: These nodes must be connected by means of the path visibleDecl("x") defined in figure 3, which connects any Exp node to the nearest enclosing declaration with name "x".

(3) *Subgraph replacement*: Erase the Exp node, create a new ApplId node and an additional ApplToDecl edge from the new node to the old DeclId node.

(4) *Embedding transformation*: Redirect any incoming edge with label 1stChild, 2ndChild, etc. from the erased node to the new node.

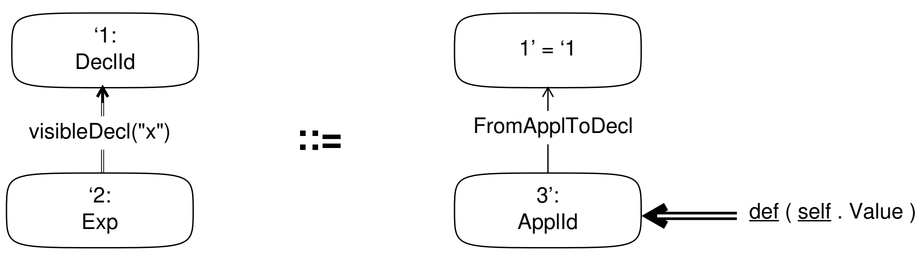
(5) *Attribute transfer*: Assign the value "x" to the Name attribute of the new node (which is not a derived but a so-called intrinsic attribute).

(6) *Attribute reevaluation*: Compute the new values of affected derived Value attributes at nodes v1, v4, and v12 in figure 2.

(7) *Check postconditions*: After the application of the rewrite rule, the new node must have a defined Value attribute[8] .

The formal representation of this production is presented in the following example:

<u>production</u> ExpandToXIdent
=



<u>embedding</u> <u>redirect</u> <-1stChild-, <-2ndChild-, <-3rdChild- <u>from</u> ‘2 <u>to</u> 3’;
<u>transfer</u> 3’.Name := "x";
<u>end</u>; *(* ExpandToXIdent *)*

Fig. 5: A typical PROGRES production with a complex application condition

Example 5.2 The Graph Rewrite Rule "ExpandToXIdent".

Let p := (AL, L, R, AR) with:

(1) AL := { visibleDecl("x", '1, '2) } .

(2) L := { *type*('1, DeclId), *type*('2, Exp),
rel(w1, 1stChild, '2), *rel*(w2, 2ndChild, '2), *rel*(w3, 3rdChild, '2) } .

(3) R := { *type*('1, DeclId), *type*(3', ApplId), *rel*(3', FromApplToDecl, '1),
rel(w1, 1stChild, 3'), *rel*(w2, 2ndChild, 3'), *rel*(w3, 3rdChild, 3'),
rel(3', Name, "x") } .

(4) AR := { ∃ v: *rel*(3', Value, v) } . ■

Note that w1, w2, and w3 are *node set identifiers*. They represent all three redirect rules within our production's embedding part. During the application of the rewrite rule, they will be mapped onto an arbitrarily large maximal and sometimes even empty sets of node identifiers in the host graph. Furthermore, the "encoding" of our production as a structure rewrite rule assumes that any selected Exp node is not source or target of edges with labels different from 1stChild etc. Otherwise, the application of the rule would attempt to produce a graph with *dangling edges* and finally fail[9]. To overcome this problem we have to add additional formulas for each edge type r to the rule's left-hand side which cause the deletion of all potentially existing r edges at the node we have to delete. These formulas have the following form:

$$rel(\text{'}2, r, w_{rout}) \quad \text{and} \quad rel(w_{rin}, r, \text{'}2)$$

with w_{rout} and w_{rin} being set identifiers. In this way, we are able to overcome the problem of the algebraic double pushout approach with deletion of nodes with unknown context.

Definition 5.3 Scheme Preserving Structure Rewriting.

S := (Φ, $\mathcal{C}$) ∈ $\mathcal{S}$(Σ) and G, G' ∈ $\mathcal{G}$(S). Furthermore, p := (AL, L, R, AR) ∈ $\mathcal{P}$(Σ). The structure G' is *derivable* from G applying p (write: G ~ p ~> G') ⇔

(1) There is a morphism u: L $\tilde{\to}$ G with: $L \subseteq_{u,AL} G$,
i.e. the via u selected redex in G respects the preconditions AL.

(2) There is no morphism $\hat{u}$: L $\tilde{\to}$ G with: $L \subseteq_{\hat{u},AL} G$ and $u \subset \hat{u}$,
i.e. u selects a maximal substructure in G (with ⊂ being the inclusion for relations).

(3) There is a morphism w: R $\tilde{\to}$ G' with: $R \subseteq_{w,AR} G'$,
i.e. the via w selected subgraph in G' respects the postconditions AR.

(4) The morphism w maps any new node identifier of R, which is not defined in L, onto a separate new node identifier in G' , which is not defined in G.

(5) With K := L ∩ R the following property holds:
v := { (x, y) ∈ u | x is identifier in K } = { (x, y) ∈ w | x is identifier in K } ,
i.e. u and w are identical with respect to the identifiers in the "gluing" structure K.

(6) There exists a graph H ∈ $\mathcal{G}$(Σ): G \ (u*(L) \ v*(K)) = H = G' \ (w*(R) \ v*(K)) ,
i.e. the intermediate result H is not required to be scheme consistent. ■

---

8) The postcondition has been added for demonstration purposes only and may be replaced by a corresponding precondition. Up to now, it is an open question which postconditions may be replaced by equivalent preconditions and postconditions have mainly been added for symmetry reasons.

9) Note that prohibition of dangling references is not part of definition 2.8 but must be enforced by appropriate formulas within a graph's scheme.

Note that this definition of structure rewriting does not prohibit the selection of *homomorphic redices* with two identifiers o1 and o2 in L being mapped onto the same object in G, even if o1 belongs to R and o2 not. These *deleting/preserving conflicts* are resolved in favor of preserving objects (unlike to the SPO approach in [18]). Otherwise, relation v, the restriction of u to K, would no longer be a morphism from K to H. For readability reasons, these forms of redex selections may be prohibited and are prohibited in the special form of graph (structure) rewriting used within the language PROGRES.

It is a straightforward task to transform the definition of scheme preserving structure rewriting above into an *effective procedure* for the application of a rule p to a scheme consistent structure G (if consistency constraints in structure schemes as well as pre- and postconditions of p are formulas the verification of which is always possible):

(1) Find a morphism u which selects a maximal redex in G for L w.r.t. preconditions AL.
(2) Build the "gluing" structure K as an intersection of L and R.
(3) Construct the intermediate structure H := G \ (u*(L) \ v*(K)) with v being the restriction of u onto the remaining object (set) identifiers in K.
(4) Then extend v to a relation w such that every new object identifier in R will be mapped onto a new object identifier which is neither defined in G nor in H.
(5) Thus we are able to define G' := H ∪ (w*(R) \ v*(K)).
(6) Finally, check whether G' is scheme consistent and respects postconditions AR.

It is easy to see that the construction above produces a structure G' such that

G ~ p ~> G' .

Note that the reevaluation of derived object properties, which are implicitly defined by sets of formulas within a scheme S, does not require explicit structure modifications. Furthermore, the steps (1) through (6) are equivalent to the construction of the diagram in figure 6, the existence of which will be proved now:

Proposition 5.4 Structure Rewriting is Construction of Diagrams.

Assuming the terminology of definition 5.3 and with G ~ p ~> G' we are able to construct the diagram of fig. 6 (with i_1 , ... , i_8 being inclusions) which has commuting subdiagrams (1) through (4).

Proof:

The existence of commuting subdiagrams (1) and (4) is guaranteed by definition 4.7. For proving the existence of commuting subdiagrams (2) and (3), we will show that step (5) above constructs a morphism v from K to H:

$$\begin{tikzcd}
AL \cup L \arrow[d, "u"'] \arrow[r, phantom, "(1)"] & L \arrow[l, "i_1"'] \arrow[d, "u"'] \arrow[r, phantom, "(2)"] & K \arrow[l, "i_2"'] \arrow[r, "i_3"] \arrow[d, "v"] \arrow[r, phantom, "(3)"] & R \arrow[r, "i_4"] \arrow[d, "w"] \arrow[r, phantom, "(4)"] & R \cup AR \arrow[d, "w"] \\
\mathcal{C}(G) \cup \Phi & G \arrow[l, "i_5"'] & H \arrow[l, "i_6"'] \arrow[r, "i_7"] & G' \arrow[r, "i_8"] & \mathcal{C}(G') \cup \Phi
\end{tikzcd}$$

Fig. 6: Fig. 6: Diagram for the application of a structure rewrite rule

To prove this, we start with definition 5.3, condition (1):

	$u: L \overset{\sim}{\to} G$
$\Leftrightarrow$ def. 4.3, prop. 4.6, and $K \subseteq L$	$u^*(K) \subseteq u^*(L) \subseteq G$
$\Rightarrow$ condition (5) of def. 5.3	$v^*(K) \subseteq u^*(L) \subseteq G$
$\Rightarrow$ simple transformation	$v^*(K) \setminus (u^*(L) \setminus v^*(K)) \subseteq G \setminus (u^*(L) \setminus v^*(K))$
$\Leftrightarrow$ simple transformation	$v^*(K) \subseteq G \setminus (u^*(L) \setminus v^*(K))$
$\Leftrightarrow$ condition (6) of def. 5.3	$v^*(K) \subseteq H$
$\Leftrightarrow$ def. 4.3 and prop. 4.6	$v: K \overset{\sim}{\to} H$.

The rest of the proof follows directly from $i_2 \circ u = v \circ i_6$ and $v \circ i_7 = i_3 \circ w$, because v is a restriction of u or w onto the identifiers in K. ■

6. Recursively Defined Structure Rewriting Transactions

So far we are able to define sets of scheme consistent rewrite rules and structure languages which are generated by applying these rewrite rules in any order to a given structure (axiom). This might be sufficient from a theoretical point of view, but when we are using structure rewriting for specification purposes additional means are necessary for *regulating the application of rewrite rules*. Although many quite different proposals for controlling application of rewrite rules to knowledge bases may be found in literature [6], the idea of using programs for this purpose, i.e. imperative control structures, seems to be the most popular one. Furthermore, so-called programmed graph grammars or programmed graph rewriting systems have already been suggested in [3, 23] and are the fundamental concept of specification languages like PAGG [13] and PROGRES [34].

On the following pages we will demonstrate that it is possible to develop a *denotational semantic*s for recursively defined, partial, and nondeterministic structure rewriting programs. These programs are termed "*transactions*" due to their atomic and consistency preserving character: any transaction applied to a scheme consistent structure either succeeds and returns another scheme consistent structure or fails without any structure modifying effects. To achieve our goal, we will first introduce a small set of *basic control flow operators*, termed *BCF operators*. By means of these operators we are able to specify any kind of structure modifying process directly or to define the semantics of more application-oriented control structures, as for instance those offered in the language PROGRES.

```
<Transaction> ::=     <TransactionId> "=" <BCF_Exp> ;
<BCF_Exp> ::=         <BasicAction> | <ActionCall> | <BCF_Term> ;
<BasicAction> ::=     "skip" | "loop" ;
<ActionCall> ::=      <RuleId> | <TransactionId> ;
<BCF_Term> ::=        "def" "(" <BCF_Exp> ")" |
                      "undef" "(" <BCF_Exp> ")" |
                      "(" <BCF_Exp> ";" <BCF_Exp> ")" |
                      "(" <BCF_Exp> "[]" <BCF_Exp> ")" ;
```

Fig. 7: : Syntax of transactions and BCF expressions

Figure 7 contains the definition of transactions as named *BCF expressions*. It distinguishes between *basic actions* like

- skip, which represents the always successful identity operator, and
- loop, which neither succeeds nor terminates for any given structure and represents therefore "crashing" or forever looping computations,

calls of basic rewrite rules and other transactions, and finally between two unary and two binary BCF operators with

- def(A) being an action which succeeds for a given structure G, whenever A applied to G produces a defined result, and returns G itself,
- undef(A) being an action which is the complement of "def(A)" and succeeds for a given structure G, whenever A applied to G terminates with failure, and returns G itself,
- (A ; B) being an action which is the sequential composition of A and B, i.e. applies first A to a given structure G and then B to any "suitable" result of the application of B, and
- (A [] B) being an action which represents the nondeterministic choice between the application of A or B to a given structure G.

Having presented the selected set of BCF operators, we are now prepared to discuss the intricacies of their intended semantics. A first problem comes with the definition of the meaning of "(A ; B)" as "apply B to *any suitable* result of A". Let us assume that the application of a transaction A to a structure G has three possible results named G1, G2, and G3, respectively. Furthermore, let us assume that the transaction B applied to G1 fails but applied to G2 and G3 succeeds. In this case we may either select G2 or G3 but not G1 as a suitable result of the application of A. This means that we need knowledge about future states of an ongoing transformation process in order to be able to discard those possible results of a single transformation step, which cause failure of the overall transformation process. It should be quite obvious that, in general, this kind of clairvoyant nondeterminism requires at least a *depth-first search implementation with backtracking* out of "dead-ends".

Another problem comes with the definition of expressions like "(A [] B)" where A loops forever applied to a certain structure G but B has a well-defined set of possible results. Having a depth-first search semantics in mind, we are forced to define the outcome of the expression "(A [] B)" as being either a nonterminating computation or any defined result produced by B. This means that the kind of *nondeterminism* we are going to define is not "angelic" but more or less "*erratic*": Using backtracking we are able to discard nondeterministic selections which lead to defined failures of basic actions or rewrite rules but not selections which cause nonterminating computations.

After an informal introduction of transactions as named BCF expressions we will now define their intended semantics by using a special form of the *fixpoint theorem* presented in [28]. In order to be able to deal with nonterminating computations we have to extend the semantic domain of binary relations over scheme consistent structures:

Definition 6.1 Extended Semantic Domain

With $\mathcal{G}(S)$ being a S-consistent class of structures for a scheme $S \in \mathcal{S}(\Sigma)$, the semantic domain of transactions is defined to be the following power set of binary relations:

$$\mathcal{D} := 2^{\mathcal{G}(S) \times (\mathcal{G}(S) \cup \{ \infty \})} .$$ ■

The semantics of a transaction is a binary relation between structures, where the symbol "∞" in a second component represents *potentially nonterminating computations*. The word "potential" includes computations with partially unknown effects, i.e. computations which either return still unknown results, or abort, or loop forever. A relation R_∞ for instance, which maps any structure G onto "∞", represents a computation with completely unknown outcomes.

In order to be able to apply fixpoint theory to recursively defined transactions we have to construct a *suitable partial order* for our semantic domain $\mathcal{D}$. "Suitable" means from a practical point of view that the relation R_∞ defined above should be less than any other element in $\mathcal{D}$ and that "R_1 less than R_2" means that R_2 is a better approximation of a given transaction than R_1. And "suitable" means from a theoretical point of view that we have to proof that any chain in $\mathcal{D}$ has a join, i.e. that any sequence of elements (R^α) with R^α being less equal than $R^{\alpha+\beta}$ for any ordinals α and β has a least upper element in $\mathcal{D}$.

Definition 6.2 Partial Order on Semantic Domain

With $R, R' \in \mathcal{D}$ our partial order "≤" is defined as follows:

$$\begin{aligned} R \le R' :\Leftrightarrow \forall\, G, G' \in \mathcal{G}:\ & (G, G') \in R \ \Rightarrow\ (\, G' = \infty \vee (G, G') \in R' \,) \\ \wedge\ & (G, \infty) \notin R \ \Rightarrow\ (\, (G, G') \in R \ \Leftrightarrow\ (G, G') \in R' \,)\,. \quad \blacksquare \end{aligned}$$

The relation R of the definition above *approximates* R' such that any input G is either related to the same set of outputs by R and R' or is related to the symbol "∞" in R and to a potentially greater set of outputs in R' (by eventually dropping "∞"). It is obvious that "≤" is indeed a partial order but we have to proof its "chains have joins" condition:

Lemma 6.3 Chains Have Joins

Let $(R^\alpha)_{\alpha \in \text{ordinal}} \subseteq \mathcal{D}$ be a chain, i.e. for any ordinals α and β holds:

$$R^\alpha \le R^{\alpha+\beta}\,.$$

Then the join of the given chain is defined as follows:

$$\begin{aligned} R := \cup(R^\alpha) := \{\, (G, G') \mid\ & G' \neq \infty\ \wedge \exists\, \alpha: (G, G') \in R^\alpha \\ \vee\ & G' = \infty\ \wedge \forall\, \alpha: (G, \infty) \in R^\alpha \,\}\,. \end{aligned}$$

Proof:

We have to show that the element R above is indeed greater than any element of our chain and, furthermore, that R is the least element in $\mathcal{D}$ with this property:

$\forall\, \alpha: R^\alpha \le R$:

$$\begin{aligned} (G, G') \in R^\alpha \ &\Rightarrow\ (G, G') \in R\ \vee\ G' = \infty\,. \\ \text{And}\ \ (G, \infty) \notin R^\alpha \ &\Rightarrow\ \forall\, \beta \ge \alpha : (G, G') \in R^\alpha \ \Leftrightarrow\ (G, G') \in R^\beta \\ &\Rightarrow\ (G, G') \in R^\alpha \ \Leftrightarrow\ (G, G') \in R\,. \end{aligned}$$

$\forall\, R' \in \mathcal{D}: (\forall\, \alpha: R^\alpha \le R') \ \Rightarrow\ R \le R'$:

$$\begin{aligned} (G, G') \in R \ &\Rightarrow\ \exists\, R^\alpha : (G, G') \in R^\alpha\ \vee\ G' = \infty \\ &\Rightarrow\ (G, G') \in R'\ \vee\ G' = \infty\,. \\ \text{And}\ \ (G, \infty) \notin R \ &\Rightarrow\ \exists\, R^\alpha : (G, \infty) \notin R^\alpha \\ &\Rightarrow\ \exists\, R^\alpha : (G, G') \in R \Leftrightarrow (G, G') \in R^\alpha \ \Leftrightarrow\ (G, G') \in R'\,. \blacksquare \end{aligned}$$

Now, we are ready to define a semantic function $\mathcal{R}$ from the syntactic domain of BCF expressions or transactions onto the semantic domain $\mathcal{D}$ inductively:

Definition 6.4 Semantics of BCF Expressions

Let A, B $\in$ <BCF_Exp>. The semantic function $\mathcal{R}$: <BCF_Exp> $\rightarrow \mathcal{D}$ is defined as follows based on definition 5.3 for a set of rewrite rules (productions) $\mathcal{P}(\Sigma)$ and for a set of S-scheme consistent structures $\mathcal{G}(S)$:

(1) $(G, G') \in \mathcal{R}[\![\underline{\text{skip}}]\!] :\Leftrightarrow \quad G = G'$.

(2) $(G, G') \in \mathcal{R}[\![\underline{\text{halt}}]\!] :\Leftrightarrow \quad G' = \infty$.

(3) $(G, G') \in \mathcal{R}[\![p]\!] :\Leftrightarrow \quad G \sim p \sim\!> G'$, for any production $p \in \mathcal{P}(\Sigma)$.

(4) $(G, G') \in \mathcal{R}[\![\underline{\text{def}}(A)]\!] :\Leftrightarrow \quad \exists \; G'' \neq \infty : \; (G, G'') \in \mathcal{R}[\![A]\!] \wedge G = G'$
$\qquad \vee \; (G, \infty) \in \mathcal{R}[\![A]\!] \wedge G' = \infty$.

(5) $(G, G') \in \mathcal{R}[\![\underline{\text{undef}}(A)]\!] :\Leftrightarrow \quad (\neg \exists \, G'': (G, G'') \in \mathcal{R}[\![A]\!]) \wedge G = G'$
$\qquad \vee \; (G, \infty) \in \mathcal{R}[\![A]\!] \wedge G' = \infty$.

(6) $(G, G') \in \mathcal{R}[\![(A ; B)]\!] :\Leftrightarrow \quad \exists \, G'' \neq \infty : (G, G'') \in \mathcal{R}[\![A]\!] \wedge (G'', G') \in \mathcal{R}[\![B]\!]$
$\qquad \vee \; (G, \infty) \in \mathcal{R}[\![A]\!] \wedge G' = \infty$.

(7) $(G, G') \in \mathcal{R}[\![(A \; [] \; B)]\!] :\Leftrightarrow \quad (G, G') \in \mathcal{R}[\![A]\!] \vee (G, G') \in \mathcal{R}[\![B]\!]$. ■

Based on these definitions we are now able to develop a fixpoint semantics for recursively defined transactions like

$$T = ((\text{OptimizeExp} ; T) \; [] \; \underline{\text{undef}}(\text{OptimizeExp})) ,$$

which is expected to apply a given optimization rule to an “Exp” graph as long as possible. For this purpose, we will use the following version of a fixpoint theorem (taken from [28]):

Proposition 6.5 Fixpoint Theorem

Let f be a monotonic function(al) on a partially ordered set in which every chain has a join, and let f^{α}, for ordinal α, be defined inductively by

$$f^{\alpha} = (\cup_{\beta \,:\, \beta < \alpha} : f(f^{\beta})) .$$

Then f has a least fixpoint given by f^{α}, for some ordinal α.

Proof:

See appendix of [28]. ■

In order to be able to use theorem 6.5 we have still to prove that BCF expressions correspond to monotonic functionals. A BCF expression which contains for instance applied transaction identifiers T_1 to T_n, must be interpreted as a functional with the following signature:

$$\mathcal{R}[\![E]\!] : \mathcal{D}^n \rightarrow \mathcal{D} .$$

Provided with the semantics $\mathcal{R}[\![T_1]\!], \ldots, \mathcal{R}[\![T_n]\!]$ of T_1 to T_n the semantics of E is given by definition 6.4 and is denoted as follows:

$$\mathcal{R}[\![E]\!] \, [\mathcal{R}[\![T_1]\!], \ldots, \mathcal{R}[\![T_n]\!]] .$$

In such a way we are able to define the semantics of all BCF expressions and transactions in the absence of recursion. But having a recursively defined transaction like

$$T = ((\text{OptimizeExp} ; T) \; [] \; \underline{\text{undef}}(\text{OptimizeExp}))$$

we are looking for a least element $R \in \mathcal{D}$ such that the following fixpoint equation

$$\mathcal{R}[\![T]\!] := R = \mathcal{R}[\![((\text{OptimizeExp} ; T) \; [] \; \underline{\text{undef}}(\text{OptimizeExp}))]\!] \, [R]$$

holds. Theorem 6.5 provides us with such a least fixpoint if we are able to show that all BCF operators define monotonic functionals and, therefore, all complex BCF expressions, too:

Lemma 6.6 BCF Operators are Monotonic Functionals

With $\mathcal{D}$ being the semantic domain for a class $\mathcal{G}(S)$ of S-scheme consistent structures, $\mathcal{R}$: <BCF_Exp> $\rightarrow \mathcal{D}$, and A, A', B $\in$ <BCF_Exp> the following holds:
$\mathcal{R}[\![A]\!] \leq \mathcal{R}[\![A']\!]$ implies:

(1) $\mathcal{R}[\![\underline{\text{def}}(A)]\!] \leq \mathcal{R}[\![\underline{\text{def}}(A')]\!]$.

(2) $\mathcal{R}[\![\underline{\text{undef}}(A)]\!] \leq \mathcal{R}[\![\underline{\text{undef}}(A')]\!]$.

(3) $\mathcal{R}[\![(A\ ;B)]\!] \leq \mathcal{R}[\![(A'\ ;B)]\!]$.

(4) $\mathcal{R}[\![(B\ ;A)]\!] \leq \mathcal{R}[\![(B\ ;A')]\!]$.

(5) $\mathcal{R}[\![(A\ [] \ B)]\!] = \mathcal{R}[\![(B\ []\ A)]\!] \leq \mathcal{R}[\![(B\ []\ A')]\!] = \mathcal{R}[\![(A'\ []\ B)]\!]$.

Proof:

ad (1): $\mathcal{R}[\![\underline{\text{def}}(A)]\!] \leq \mathcal{R}[\![\underline{\text{def}}(A')]\!]$:

$G \in \mathcal{G}(S)$ with $(G, \infty) \notin \mathcal{R}[\![A]\!]$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![A]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![A']\!]$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![\underline{\text{def}}(A)]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![\underline{\text{def}}(A')]\!]$.

$G \in \mathcal{G}(S)$ with $(G, \infty) \in \mathcal{R}[\![A]\!]$
$\Rightarrow \quad (G, \infty) \in \mathcal{R}[\![\underline{\text{def}}(A)]\!]$
and with $(G, G') \in \mathcal{R}[\![A]\!] \Rightarrow (G, G') \in \mathcal{R}[\![A']\!] \vee G' = \infty$:
$\quad (G, G') \in \mathcal{R}[\![\underline{\text{def}}(A)]\!]$
$\Rightarrow \quad (\exists G'' \neq \infty : (G, G'') \in \mathcal{R}[\![A]\!] \wedge G = G') \vee G' = \infty$
$\Rightarrow \quad (\exists G'' \neq \infty : (G, G'') \in \mathcal{R}[\![A']\!] \wedge G = G') \vee G' = \infty$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![\underline{\text{def}}(A')]\!] \vee G' = \infty$.

ad (2): $\mathcal{R}[\![\underline{\text{undef}}(A)]\!] \leq \mathcal{R}[\![\underline{\text{undef}}(A')]\!]$:

$G \in \mathcal{G}(S)$ with $(G, \infty) \notin \mathcal{R}[\![A]\!]$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![A]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![A']\!]$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![\underline{\text{undef}}(A)]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![\underline{\text{undef}}(A')]\!]$.

$G \in \mathcal{G}(S)$ with $(G, \infty) \in \mathcal{R}[\![A]\!]$ implies:
$\quad (G, G') \in \mathcal{R}[\![\underline{\text{undef}}(A)]\!]$
$\Leftrightarrow \quad G' = \infty$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![\underline{\text{undef}}(A')]\!] \vee G' = \infty$.

ad (3): $\mathcal{R}[\![(A\ ;B)]\!] \leq \mathcal{R}[\![(A'\ ;B)]\!]$:

$G \in \mathcal{G}(S)$ with $(G, \infty) \notin \mathcal{R}[\![A]\!]$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![A]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![A']\!]$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![(A\ ;B)]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![(A'\ ;B)]\!]$.

$G \in \mathcal{G}(S)$ with $(G, \infty) \in \mathcal{R}[\![A]\!]$
$\Rightarrow \quad (G, \infty) \in \mathcal{R}[\![(A\ ;B)]\!]$
and with $(G, G') \in \mathcal{R}[\![A]\!] \Rightarrow (G, G') \in \mathcal{R}[\![A']\!] \vee G' = \infty$:
$\quad (G, G') \in \mathcal{R}[\![(A\ ;B)]\!]$
$\Rightarrow \quad (\exists G'' \neq \infty : (G, G'') \in \mathcal{R}[\![A]\!] \wedge (G'', G') \in \mathcal{R}[\![B]\!]) \vee G' = \infty$
$\Rightarrow \quad (\exists G'' \neq \infty : (G, G'') \in \mathcal{R}[\![A']\!] \wedge (G'', G') \in \mathcal{R}[\![B]\!]) \vee G' = \infty$
$\Rightarrow \quad (G, G') \in \mathcal{R}[\![(A'\ ;B)]\!] \vee G' = \infty$.

ad (4): $\mathcal{R}[\![(B ; A)]\!] \leq \mathcal{R}[\![(B ; A')]\!]$:

$G \in \mathcal{G}(S)$ with $\neg \exists G'' : ((G, G'') \in \mathcal{R}[\![B]\!] \wedge (G'', \infty) \in \mathcal{R}[\![A]\!])$
and with any $G'' \in \mathcal{G}(S)$:

$(G, G'') \in \mathcal{R}[\![B]\!]$
$\Rightarrow (G'', \infty) \notin \mathcal{R}[\![A]\!]$
$\Rightarrow (G'', G') \in \mathcal{R}[\![A]\!] \Leftrightarrow (G'', G') \in \mathcal{R}[\![A']\!]$
$\Rightarrow (G, G') \in \mathcal{R}[\![(B ; A)]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![(B ; A')]\!]$.

$G \in \mathcal{G}(S)$ with $\exists G'' : ((G, G'') \in \mathcal{R}[\![B]\!] \wedge (G'', \infty) \in \mathcal{R}[\![A]\!])$
$\Rightarrow (G, \infty) \in \mathcal{R}[\![(B ; A)]\!]$
and with $(G'', G') \in \mathcal{R}[\![A]\!] \Rightarrow (G'', G') \in \mathcal{R}[\![A']\!] \vee G' = \infty$:

$(G, G') \in \mathcal{R}[\![(B ; A)]\!]$
$\Rightarrow (\exists G'' \neq \infty : (G, G'') \in \mathcal{R}[\![B]\!] \wedge (G'', G') \in \mathcal{R}[\![A]\!]) \vee G' = \infty$
$\Rightarrow (\exists G'' \neq \infty : (G, G'') \in \mathcal{R}[\![B]\!] \wedge (G'', G') \in \mathcal{R}[\![A']\!]) \vee G' = \infty$
$\Rightarrow \mathcal{R}[\![(B ; A')]\!] \vee G' = \infty$.

ad (5): $\mathcal{R}[\![(A [] B)]\!] = \mathcal{R}[\![(B [] A)]\!] \leq \mathcal{R}[\![(B [] A')]\!] = \mathcal{R}[\![(A' [] B)]\!]$:

$G \in \mathcal{G}(S)$ with $(G, \infty) \notin \mathcal{R}[\![A]\!]$
$\Rightarrow (G, G') \in \mathcal{R}[\![A]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![A']\!]$
$\Rightarrow (G, G') \in \mathcal{R}[\![(A [] B)]\!] \Leftrightarrow (G, G') \in \mathcal{R}[\![(A' [] B)]\!]$.

$G \in \mathcal{G}(S)$ with $(G, \infty) \in \mathcal{R}[\![A]\!]$
$\Rightarrow (G, \infty) \in \mathcal{R}[\![(A [] B)]\!]$
and with $(G, G') \in \mathcal{R}[\![A]\!] \Rightarrow (G, G') \in \mathcal{R}[\![A']\!] \vee G' = \infty$:

$(G, G') \in \mathcal{R}[\![(A [] B)]\!]$
$\Rightarrow (G, G') \in \mathcal{R}[\![A]\!] \vee (G, G') \in \mathcal{R}[\![B]\!]$
$\Rightarrow (G, G') \in \mathcal{R}[\![A']\!] \vee (G, G') \in \mathcal{R}[\![B]\!] \vee G' = \infty$
$\Rightarrow (G, G') \in \mathcal{R}[\![(A' [] B)]\!] \vee G' = \infty$. ■

Corollary 6.7 Fixpoint Semantics for Transactions

Let T_1 to T_n be a set of transactions which contain in their bodies E_1 to E_n at most calls to transactions T_1 through T_n , i.e. $T_1 = E_1[T_1, \ldots, T_n], \ldots, T_n = E_n[T_1, \ldots, T_n]$. With $\mathcal{G}(S)$ being the corresponding class of scheme consistent structures, the following holds:

(1) $T_1, \ldots, T_n$ have least fixpoints $\mathcal{R}[\![T_1]\!], \ldots, \mathcal{R}[\![T_n]\!] \in \mathcal{D}$.

(2) Approximations of these fixpoints may be constructed in the following way:
$R_1^0 := \mathcal{G}(S) \times \{ \infty \} ; \ldots ; R_n^0 := \mathcal{G}(S) \times \{ \infty \}$,
$R_1^{k+1} := \mathcal{R}[\![E_1]\!] [R_1^k, \ldots, R_n^k] ; \ldots ; R_n^{k+1} := \mathcal{R}[\![E_1]\!] [R_1^k, \ldots, R_n^k]$,
where $\mathcal{R}[\![E_i]\!] : \mathcal{D}^n \to \mathcal{D}$ takes approximations for $T_1, \ldots, T_n$ and yields a new approximation for T_i by applying definition 6.4 to expression E_i .

Proof:

ad (1): Follows directly from theorem 6.5 and lemmata 6.3 and 6.6.

ad (2): The relations R_i^k are indeed approximations for $T_1, \ldots, T_n$ such that for any i, k: $R_i^0 \leq \ldots \leq R_i^k \leq \mathcal{R}[\![T_i]\!]$. This follows directly from theorem 6.5 and lemmata 6.3 and 6.6 with

$$\forall i, l : R_i^l = \cup_{k \leq l} R_i^k \leq \mathcal{R}[\![T_i]\!] \text{ and } \cup_{k \in \{\}} R_i^k = \mathcal{G}(S) \times \{ \infty \} . ■$$

Note that the corollary above is not only interesting from a theoretical point of view by guaranteeing the existence of least fixpoints for any recursively defined set of transactions. Additionally, it provides us with a *straightforward algorithm* which computes the results of terminating transactions and approximates the results of (potentially) nonterminating transactions.

7. Conclusion

The definitions and propositions of the preceding sections constitute a *new logic based framework* for the formal treatment of programmed graph rewriting systems as special cases of programmed structure rewriting systems. The new calculus contains the following graph or structure rewriting approaches as special cases:

- the algebraic double pushout approach of [7, 9] and its generalization in the form of so-called structure grammars, a special case of high-level replacement systems, in [8],
- the programmed graph grammar approach of [13] with the exception of certain forms of embedding rules which consist of very complex graph patterns[10],
- the expression-oriented graph grammar approach of [24] and the structured graph grammar approach of [16] without any restrictions,
- and finally the "programmed derivation of relational structures" approach in [22] (if we omit the "pseudo parallel" interleaving operator "$\|$" proposed there).

Furthermore, our formalism tries to close the gap between the "operation-oriented" manipulation of data structures by means of rewrite rules and the "declaration-oriented" description of data structures by means of logic based knowledge representation languages. In this way, both disciplines - graph grammar theory and mathematical logic theory - might be able to profit from each other:

- Structure (graph) rewrite rules might be a very comfortable and well-defined mechanism for manipulating knowledge bases or deductive data bases; cf. [12, 32], whereas
- many logic based techniques have been proposed for efficiently maintaining derived properties of data structures, solving constraint systems, and even for proving the correctness of certain kinds of data manipulations (cf. [2, 15, 30, 38]).

Last but not least the new approach provides us with a *formal definition of recursive structure rewriting programs*, termed transactions. By adding a new symbol "∞" to the domain of structures, which represents unknown results, we were able to overcome the difficulties discussed in [35]. There, control graphs instead of BCF expressions are used to program with graph rewrite rules[11]. These control graphs are similar to those already presented in [23], but additionally support testing whether the execution of a complex graph transformation to a given graph would be successful or not. This has the unwanted consequence that the semantic function from the domain of control graphs onto the range of binary graph relations in [35] is partial, i.e. there exist control graphs without any interpretation as binary relations.

---

10) Embedding based on complex graph patterns requires the usage of sets of tuples of object identifiers instead of sets of object identifiers only.

11) Note that control graphs are slightly more general than BCF expression; they support the abolished style of "goto" programming, whereas BCF expressions support well-structured programming only.

Finally note that [28], which presents an *axiomatic semantics definition* for nondeterministic partial commands, gave us the initial impulse for the introduction of the symbol "∞" as well as for the selection of the adequate fixpoint theorem version. Nevertheless, we had to prefer the "denotational" instead of the "axiomatic" approach, because structure rewrite rules play here about the same role as simple assignments in [28]. The definition of their intended semantics as binary relations is given in definition 5.3, but it is a yet unsolved problem how to "push" postconditions through rewrite rules for obtaining their corresponding weakest preconditions (see question (3) below).

Being the first attempt to present a logic-based framework for a very general form of rewriting systems there remain a number of *open questions*:

(1) For which subset of our new approach are subdiagrams (2) and (3) of fig. 6 pushouts beyond those subcases identified in [8]?

(2) Can we characterize a "useful" subset of rewrite rules and consistency constraints such that an effective proof procedure exists for the "are all derivable structures scheme consistent" problem which has already considered in [5] for restricted forms of graph grammars and monadic second order logic formulas?

(3) Can we develop a general procedure which transforms any set of rewrite rule postconditions into weakest preconditions?

The problems (2) and (3) above are related to each other by transforming global consistency constraints into equivalent postconditions of individual rewrite rules using techniques proposed in [2]. Having such a procedure which translates postconditions into corresponding preconditions, problem (2) is reducible to the question whether these new preconditions are already derivable from the original sets of preconditions of rewrite rules.

But note that the main motivation for the development of the new formalism was not to generate these problems but to provide us with a solid fundament for a precise description of the application-oriented graph grammar language PROGRES in [35]. Therefore, we are currently spending most of our time with realizing an *integrated PROGRES programming environment*. Up to now, this environment has about 400.000 lines of code and offers a syntax-aided editor, an incrementally working type checker, a number of browsing tools, and finally an integrated interpreter as well as a cross-compiler to Modula-2 and C. A first release of these tools and the underlying graph-oriented database system GRAS are available as free software via anonymous ftp from ftp-server ftp.informatik.rwth-aachen.de in the directories pub/unix/PROGRES and pub/unix/GRAS.

References

[1] Ausiello G., Atzeni P. (eds.): *Proc. Int. Conf. on Database Theory*, LNCS 243, Berlin: Springer Verlag (1986)

[2] Bry F., Manthey R., Martens B.: *Integrity Verification in Knowledge Bases*, in: [37], 114-139

[3] Bunke H.: *Sequentielle und parallele programmierte Graphgrammatiken*, Dissertation, Technical Report IMMD-7-7, Universität Erlangen, Germany (1974)

[4] Claus V., Ehrig H., Rozenberg G.: *Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, LNCS 73, Berlin: Springer Verlag (1979)

[5] Courcelle B.: *Graphs as Relational Structures: An Algebraic and Logical Approach*, in: [10], 238-252

[6] Dassow J., Paun G.: *Regulated Rewriting in Formal Language Theory*, EATCS 18, Berlin: Springer Verlag (1989)

[7] Ehrig H., Habel A.: *Graph Grammars with Application Condition*, in: [31], 87-100

[8] Ehrig H., Habel A., Kreowski H.-J., Parisi-Presicce F.: *From Graph Grammars to High-Level Replacement Systems*, in: [10], 269-291

[9] Ehrig H.: *Introduction to the Algebraic Theory of Graph Grammars (a Survey)*, in: [4], 1-69

[10] Ehrig H., Kreowski H.-J., Rozenberg G.: *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Berlin: Springer Verlag (1991)

[11] Engels G., Lewerentz C., Nagl M., Schäfer W., Schürr A.: *Building Integrated Software Development Environments Part I: Tool Specification*, in ACM Transactions on Software Engineering and Methodology, vol. 1, no. 2, New York: acm Press (1992), 135-167

[12] Gallaire H.: *Logic and Data Bases*, New York: Plenum Press (1978)

[13] Göttler H.: *Graphgrammatiken in der Softwaretechnik*, IFB 178, Springer Verlag (1988)

[14] Große-Wienker R., Hermanns O., Menzenbach D., Pollack A., Repetzki S., Schwartz J., Sonnenschein K., Westfechtel B: *Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme*, Technical Report AIB-93-11, RWTH Aachen, Fachgruppe Informatik, Germany (1993)

[15] Van Hentenrynck P.: *Constraint Satisfaction in Logic Programming*, Cambridge: MIT Press (1989)

[16] Kreowski H.-J., Rozenberg G.: *On Structured Graph Grammars: Part I and II*, Technical Report 3/88, University of Bremen, FB Mathematik/Informatik (1988)

[17] Kiesel N., Schürr A., Westfechtel B.: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*, in: [LRJ 93], 272-286

[18] Löwe M.: *Algebraic Approach to Graph Transformation Based on Single Pushout Derivations*, TR No. 90/5, TU Berlin, Germany (1990)

[19] Lee H.-Y., Reid Th.F., Jarzabek St. (eds.): *CASE '93 Proc. 6th Int. Workshop on Computer-Aided Software Engineering*, Washington: IEEE Computer Society Press (1993)

[20] Mylopoulos J., Borgida A., Jarke M., Koubarkis M.: *Telos: a Language for Representing Knowledge about Information Systems*, ACM Transactions on Information Systems, Vol. 8, No. 4, New York: acm Press (1990), 325-362

[21] Minker J.: *Perspectives in Deductive Databases*, in: The Journal of Logic Programming, Amsterdam: Elsevier Science Publ. (1988), 33-60

[22] Maggiolo-Schettini A., Winkowski J.: *Programmed Derivations of Relational Structures*, in: [10], 582-598

[23] Nagl M.: *Graph-Grammatiken*, Braunschweig: Vieweg Press (1979)

[24] Nagl M.: *A Tutorial and Bibliographical Survey on Graph Grammars*, in: [4], 70-126

[25] Nagl M.: *Graph Technology Applied to a Software Project*, in: [31], 303-322

[26] Naqvi Sh.A.: *Some Extensions to the Closed World Assumption in Databases*, in: [1], 341-348

[27] Naqvi Sh.A.: *Negation as Failure for First-Order Queries*, in: Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, New York: acm Press (1986), 114-122

[28] Nelson G.: *A Generalization of Dijkstra's Calculus*, in ACM Transactions on Programming Languages and Systems, vol. 11, no. 4, New York: acm Press, USA (1989), 517-561

[29] Nagl M., Schürr A.: *A Specification Environment for Graph Grammars*, in: [10], 599-609

[30] Naqvi Sh.A., Tsur Sh.: *Data and Knowledge Bases*, Washington: IEEE Computer Society Press (1989)

[31] Rozenberg G., Salomaa A.: *The Book of L*, Berlin: Springer Verlag (1986)

[32] Rybinski H.: *On First-Order-Logic Databases,* in: ACM Transaction on Database Systems, Vol. 12, No.3, New York: acm Press (1987), 325-349

[33] Schmidt G., Berghammer R.: *Proc. Int. Workshop on Graph- Theoretic Concepts in Computer Science (WG '91)*, LNCS 570, Berlin: Springer Verlag(1991)

[34] Schürr A.: *PROGRES: A VHL-Language Based on Graph Grammars*, in: [10], 641-659

[35] Schürr A.: *Operationales Spezifizieren mit Programmierten Graphersetzungsystemens: Formale Definitionen, Anwendungen und Werkzeuge*, Braunschweig: Deutscher Universitätsverlag (1991)

[36] Schürr A., Zündorf A.: *Non-Deterministic Control Structures for Graph Rewriting Systems*, in: [33], 48-62

[37] Voronkov A. (ed.): *Logic Programming,* LNCS 592, Berlin: Springer Verlag (1991)

[38] Thaise A. (ed.): *From Standard Logic to Logic Programming*, New York: John Wiley & Sons Ltd. (1989)