

Logic Based Programmed Structure Rewriting Systems¹

Andy Schürr²

*Lehrstuhl für Informatik III, RWTH Aachen,
Ahornstr. 55, D-52074 Aachen, Germany
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 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 or 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 [10, 13, 24]. 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 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.

1) Technical Report AIB-94-13, RWTH Aachen; appears in: Rozenberg (ed.): Fundamenta Informaticae Special Issue on Graph Transformation Systems, Amsterdam: IOS Press

2) Partially supported by Netherland's Organization for Scientific Research (NWO) under grant B 62-427.

One result of this design process is the language **PROGRES** - an acronym for “**PRO**-grammed **G**raph **R**ewriting **S**ystem” - which possesses a well-defined syntax, static and dynamic semantics. It represents the first attempt to combine the algorithmic graph grammar approach with new elements for the declaration of graph integrity constraints as well as 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 formalize

- 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 *nonmonotonic logic and fixpoint theory* and has some similarities to underlying theories of knowledge representation languages or deductive database systems [20, 29]. To summarize, the acronym **PROGRES** is a “trade mark” for an integrated set of tools, a very high level programming or specification language, and its underlying formalism. The main purpose of this paper is to present a generalized version of **PROGRES**’ underlying formalism, a new *programmed structure rewriting approach*, whereas previously published papers had their main focus on the language **PROGRES** and its integrated programming environment [28, 34] or neglected the aspect of programming with rewrite rules [36]. As a matter of fact, the paper is a revised and considerably extended version of [36] and organized as follows:

- Section 2 repeats some basic terminology of predicate logic and contains the formal definition of structures and structure schemas as sets of formulas.
- Section 3 is dedicated to the formal definition of substructures and introduces the new structure rewriting formalism itself.
- Section 4 deals with recursively defined structure rewriting programs, so-called transactions, and their denotational semantics definition.
- Section 5 discusses the relationships between the presented rewriting approach and other (graph) rewriting formalisms.
- 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.

Within all these sections, we will use a coherent running example drawn from the area of (deductive) database systems. It is the specification of a graph-like person database including the definition of derived relationships and complex update operations (figure 1 displays a sample database).

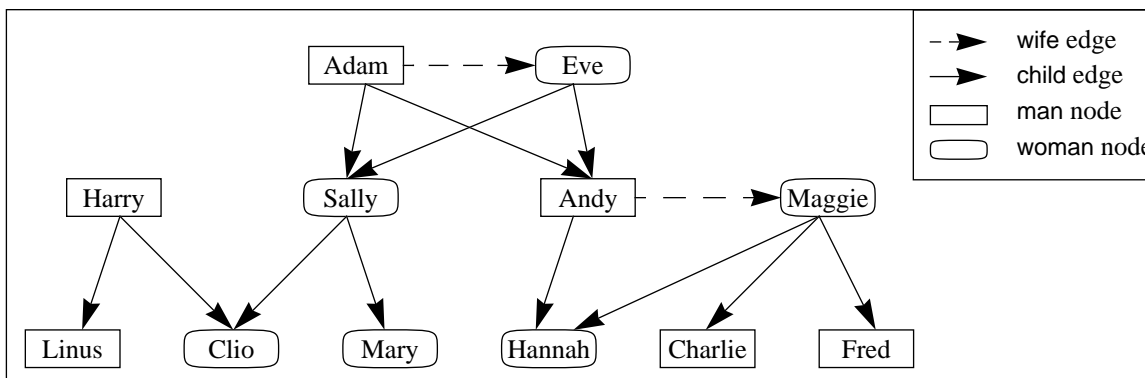


Fig. 1: The running example, a person database

2. Structure Schemas and Schema Consistent Structures

This section introduces basic terminology of predicate logic and explains the modelling of node and edge labeled graphs as special cases of *schema consistent structures*, i.e. as sets of formulas with certain properties. Afterwards, our running example of a person database will be modelled as a directed graph. In this way, we are able to demonstrate that graphs are a special case of structures and graph rewriting is a special case of structure rewriting. Nevertheless, 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 [8], and should even be adaptable to the case of almost arbitrary knowledge bases [20, 29].

Definition 2.1 Signature.

A 5-tuple $\Sigma := (\mathcal{A}_F, \mathcal{A}_P, \mathcal{V}, \mathcal{W}, \mathcal{X})$ is a **signature** iff:

- (1) \mathcal{A}_F is an alphabet of function symbols³ (including constants as a special case).
- (2) \mathcal{A}_P is an alphabet of predicate symbols⁴.
- (3) \mathcal{V} is a special alphabet of object identifiers.
- (4) \mathcal{W} is a special alphabet of identifiers for sets of objects.
- (5) \mathcal{X} is an alphabet of logical variables used for quantification purposes. ■

The signature of the following example may be used for the definition of person databases, or more precisely, their graph representations as sets of formulas. Figure 1 (on the previous page) presents such a graph representation of a database with twelve persons. Each person is either a man node or a woman node. Both man and woman nodes may be sources or targets of child edges, whereas wife edges always have a man as source and a woman as target. For reasons of simplicity, attributes of persons are neglected throughout the paper, and meaningful object identifiers are used instead of name attributes.

Example 2.2 Signature for a Person Database.

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

- (1) $\mathcal{A}_F := \{ \text{child, woman, wife, man, person, ...} \}$.
- (2) $\mathcal{A}_P := \{ \text{node, edge, ...} \}$.
- (3) $\mathcal{V} := \{ \text{He, She, ...} \}$.
- (4) $\mathcal{W} := \{ \text{HerSons, HisSons, HerDaughters, HisDaughters, ...} \}$.
- (5) $\mathcal{X} := \{ x_1, x_2, ... \}$. ■

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

- The predicate symbol “node” with “node(x, l)” for “node x has label l”, and
- the predicate symbol “edge” with “edge(x, e, y)” for “edge with label e connects source node x to target node y”.

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

3) More precisely, a family of alphabets of symbols if we take domain and range of functions into account.

4) A family of alphabets of symbols if we take arities of predicates into account.

Definition 2.3 Σ -Term and Σ -Atom.

$\mathcal{T}(\Sigma)$ is the set of all **terms** (in the usual sense) which contain function symbols and constants of \mathcal{A}_F , free variables of \mathcal{X} , and additional constant symbols of \mathcal{V} and \mathcal{W} . $\mathcal{A}(\Sigma)$ is the set of all **atomic formulas** over $\mathcal{T}(\Sigma)$ which contain predicate symbols of \mathcal{A}_P and “=” for expressing the equality of two Σ -terms. ■

Definition 2.4 Closed Σ -Formula and Derivation of Σ -Formulas.

$\mathcal{F}(\Sigma)$ is the set of all **closed first order predicate logic formulas**⁵ with $\mathcal{A}(\Sigma)$ as atomic formulas, \wedge, \vee, \dots as logical connectives, and \exists, \forall as quantifiers. Furthermore 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 and complete) inference system of first order predicate logic with equality. ■

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

Definition 2.5 Σ -Structure.

A closed set of formulas $F \in \mathcal{F}(\Sigma)$ is a **Σ -structure** (write: $F \in \mathcal{L}(\Sigma)$) \Leftrightarrow

$F \subseteq \mathcal{A}(\Sigma)$ and F does not contain formulas of the form “ $\tau_1 = \tau_2$ ”. ■

Example 2.6 A Person Database Structure.

The following structure is a set of formulas which has the graph F of fig. 1 as a model but which also has many other graphs as models, in which we are not interested in:

$F := \{ \text{node(Adam, man), node(Eve, woman), edge(Adam, Wife, Eve), ... } \}$. ■

In definition 2.7, we will introduce a so-called “completing operator”, which allows us to get rid of unwanted (graph) models of Σ -structures 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 in the sequel.

The following example demonstrates our needs for a completing operator in more detail. It presents the definition of a single person database and explains our difficulties to prove that this database contains indeed one person only. 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 by using “*nonmonotonic reasoning*” capabilities (cf. [20, 25, 26]). 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. The resulting set of formulas has to be consistent (it may not contain contradictions) and “sufficiently complete” such that we can prove the above mentioned properties by using the axioms of “pure” first-order predicate logic only.

5) Requiring bindings for all logical variables (by means of existential or universal quantification) does not restrict the expressiveness of formulas but avoids any difficulties with varying treatments of free variables in different inference systems.

Definition 2.7 Σ -Structure Completing Operator.

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

For all structures $F \in \mathcal{L}(\Sigma)$: $F \subseteq C(F)$ and $C(F)$ is a consistent set of formulas. ■

Example 2.8 Nonmonotonic Reasoning.

The singleton set $F := \{ \text{node}(\text{Adam}, \text{man}) \}$ is a structure which has all graphs containing at least one “man” node as models. Being interested in properties of “minimal” F graph models only, we should be able to prove that F contains one node. For this purpose, the operator C has to be defined as follows (omitted sets of formulas deal with edges etc.):

$$C(F) := F \cup \dots \cup$$

$$\{ \forall x, l: \text{node}(x, l) \rightarrow (x = v_1 \vee \dots \vee x = v_k) \mid v_1, \dots, v_k \in \mathcal{V} \text{ are all object ids in } F \}.$$

Now, we are able to prove that F contains only the male node Adam:

$$C(F) \vdash (\forall x, l: \text{node}(x, l) \rightarrow x = \text{Adam} \wedge l = \text{man}). \quad \blacksquare$$

Completing operators may have definitions, which are specific for a regarded class of structures. Therefore, they are part of the following definition of *structure schemas*:

Definition 2.9 Σ -Structure Schema.

A tuple $S := (\Phi, C)$ is a **Σ -structure schema** (write: $S \in \mathcal{S}(\Sigma)$) \Leftrightarrow

- (1) $\Phi \in \mathcal{F}(\Sigma)$ is a consistent set of formulas without references to specific object (set) identifiers (it contains integrity constraints and derived data definitions).
- (2) C is a Σ -structure completing operator. ■

Definition 2.10 Schema Consistent Structure.

Let $S := (\Phi, C) \in \mathcal{S}(\Sigma)$ a schema. A Σ -structure $F \in \mathcal{L}(\Sigma)$ is **schema consistent with respect to S** (write: $F \in \mathcal{L}(S)$) \Leftrightarrow

$$C(F) \cup \Phi \text{ is a consistent set of formulas.} \quad \blacksquare$$

This definition of schema consistency is very similar to a related definition of the knowledge representation language Telos [20]. It states that a structure is inconsistent with respect to a schema, if and only if we are able to derive contradicting formulas from the completed structure and its schema. The following example is the definition of a person database schema. Its set of formulas Φ consists of three subsets. A first subset defines integrity constraints for the intermediate data model of directed graphs and excludes “dangling” edges. The second subset contains all person database specific integrity constraints, like “the database contains male and female nodes only”. The last subset deals with a number of derived graph properties like the well-known ancestor relation or the definition of the brother of a child.

Example 2.11 A Schema for Person Databases.

A structure schema $S := (\Phi, C)$ for the graph F in fig. 1 has about the following form:

$$\begin{aligned} \Phi := & \{ \forall x, e, y: \text{edge}(x, e, y) \rightarrow \exists x_l, y_l: \text{node}(x, x_l) \wedge \text{node}(y, y_l), \dots \} \\ & \cup \{ \forall x, l: \text{node}(x, l) \rightarrow (l = \text{man} \vee l = \text{woman}), \\ & \quad \forall x, y,: \text{edge}(x, \text{wife}, y) \rightarrow \text{node}(x, \text{man}) \wedge \text{node}(y, \text{woman}), \\ & \quad \forall x, y, z: \text{edge}(x, \text{wife}, y) \wedge \text{edge}(x, \text{wife}, z) \rightarrow y = z, \dots \} \\ & \cup \{ \forall x, y: \text{ancestor}(x, y) \leftrightarrow (\exists z: \text{edge}(y, \text{child}, z) \wedge (z = y \vee \text{ancestor}(z, y))), \\ & \quad \forall x, y: \text{brother}(x, y) \leftrightarrow \exists z: \text{edge}(z, \text{child}, x) \wedge \text{edge}(z, \text{child}, y) \\ & \quad \wedge \text{node}(y, \text{man}) \wedge x \neq y, \dots \}. \end{aligned}$$

$$C(F) := F \cup \dots \quad \blacksquare$$

3. Schema Preserving Structure Rewriting

The first part of this section formalizes the term “redex under additional constraints” by means of morphisms. This term is central to the definition of “structure rewriting with pre- and postconditions”, which follows afterwards. It 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. The main difficulty with the definition of the new structure rewriting approach was the requirement that it should include the expression oriented algorithmic graph grammar approach [23]. This approach has very powerful embedding rules, which are able to delete, copy, and redirect arbitrary large bundles of edges.

We will handle embedding rules of algorithmic graph grammar approaches by introducing special *object set identifiers* on a structure rewrite rule’s left- and right-hand side. These set identifiers match an arbitrarily large (maximal) set of object identifiers in a given redex (see example 3.7 and definition 3.8). As a consequence, 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.

Definition 3.1 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

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

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 $w \in \mathcal{W}_F$: $u(w) \subseteq \mathcal{V}_{F'}$,
i.e. every source object set identifier will be mapped onto a set of target object identifiers.

Let u^* be the natural extension⁶ 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:

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

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 manipulate sets of arbitrary object properties (e.g. attributes of nodes).

Definition 3.2 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: $u: F \rightsquigarrow F'$) \Leftrightarrow

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

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

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

Proposition 3.3 The Category of Σ -Structures.

Assume “ \circ ” to be the usual composition of binary relations. Then, $\mathcal{F}(\Sigma)$ together with the family of Σ -morphisms defined above and “ \circ ” is a *category*; the same holds true for the set of Σ -structures $\mathcal{L}(\Sigma)$ and the family of Σ -structure morphisms.

Proof:

Σ -Morphisms are closed w.r.t. “ \circ ”, i.e. $u: F \rightsquigarrow F', u': F' \rightsquigarrow F'' \Rightarrow (u \circ u'): F \rightsquigarrow F''$:
 u and u' are Σ -relations $\Rightarrow_{\text{def. 3.1}}$ $(u \circ u')$ is a Σ -relation and $(u \circ u')^* = u^* \circ u'^*$.
 u, u' are morphisms $\Rightarrow_{\text{def. 3.2}}$ $F' \vdash u^*(F)$ and $F'' \vdash u'^*(F')$
 $F' \vdash u^*(F) \Rightarrow_{\text{renaming preserves proofs}}$ $u'^*(F') \vdash u'^*(u^*(F)) = (u \circ u')^*(F)$
 $\Rightarrow_{\text{modus ponens}}$ $F'' \vdash (u \circ u')^*(F)$,
 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,

$$\text{id}_F^*(F) = F \Rightarrow F \vdash \text{id}_F^*(F) \Rightarrow \text{id}_F: F \rightsquigarrow F ,$$

i.e. id_F is the required neutral morphism.

Associativity of “ \circ ” for Σ -morphisms:

Follows directly from the fact that “ \circ ” is associative for binary relations.

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

Definition 3.4 Substructure.

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

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

Proposition 3.5 Soundness of Substructure Property.

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

F is a substructure of F' with respect to a Σ -relation u
 $\Leftrightarrow u^*(F)$ is a subset of F' .

Proof:

$$F \subseteq_u F' \Leftrightarrow_{\text{see def. 4.5}} u: F \rightsquigarrow F' \Leftrightarrow_{\text{see def. 4.3}} F' \vdash u^*(F) \Leftrightarrow_{u^*(F), F' \in \mathcal{L}(\Sigma)} u^*(F) \subseteq F' .$$

The last step of the proof follows from the fact that F and F' are sets of atomic formulas without “=”, such that “ \subseteq ” (normal set inclusion) and “ \vdash ” are equivalent relations. ■

Definition 3.6 Substructure with Additional Constraints.

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

- (1) $F \subseteq_u F'$, i.e. $F' \vdash u^*(F)$.
- (2) $u: \Psi \cup F \rightsquigarrow \Phi \cup \mathcal{C}(F')$, i.e. $\Phi \cup \mathcal{C}(F') \vdash u^*(\Psi \cup F)$.

These conditions are equivalent to the existence of the diagram in figure 2, with inclusions i_1 and i_2 being morphisms:

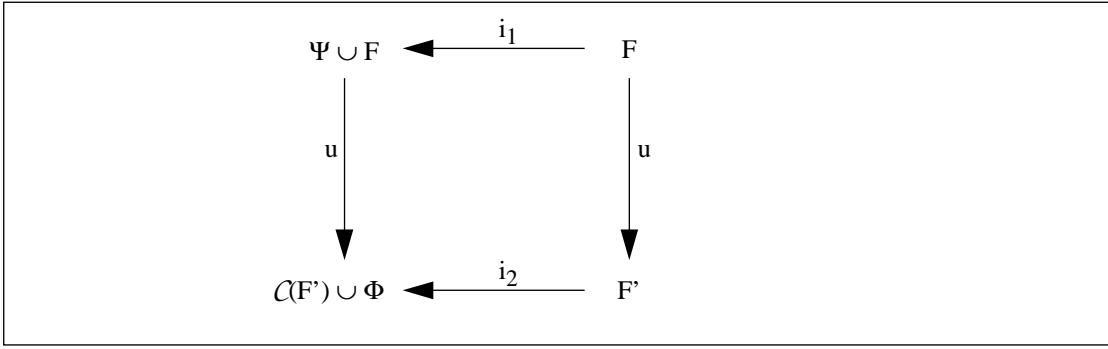


Fig. 2: Substructure (redex) selection with additional constraints

$i_1: F \xrightarrow{\sim} \Psi \cup F$, since $i_1^*(F) = F \subseteq \Psi \cup F$, i.e. $\Psi \cup F \vdash i_1^*(F)$.

$i_2: F' \xrightarrow{\sim} \Phi \cup C(F')$, since $i_2^*(F') = F' \subseteq C(F')$, i.e. $\Phi \cup C(F') \vdash i_2^*(F')$. ■

Informally speaking, a structure F is a substructure of F' with respect to additional constraints, if and only if we are able to prove that all constraints for the embedding of F in F' are fulfilled. As assumptions for this prove we may use the basic facts of F' including all formulas generated by the completing operator C , and the set of formulas Φ of the structure schema S .

Example 3.7 Substructure Selection.

Let $S := (\Phi, C)$ be the schema of example 2.11 and F' the database of example 2.6. The structure $F \in \mathcal{L}(S)$ and its accompanying set of constraints $\Psi \in \mathcal{F}(\Sigma)$, defined below, represent the query “select any pair of persons, who are allowed to marry each other, with all children”:

$F := \{ \text{node}(\text{He}, \text{man}), \text{node}(\text{She}, \text{woman}),$
 $\text{edge}(\text{He}, \text{child}, \text{HisSons}), \text{edge}(\text{He}, \text{child}, \text{HisDaughters}),$
 $\text{edge}(\text{She}, \text{child}, \text{HerSons}), \text{edge}(\text{She}, \text{child}, \text{HerDaughters}) \} .$

$\Psi := \{ \neg \text{brother}(\text{She}, \text{He}), \neg \exists x: \text{edge}(\text{He}, \text{wife}, x), \neg \exists x: \text{edge}(x, \text{wife}, \text{She}) \} .$

Remember that He and She are object identifiers, whereas HisSons, HerDaughters etc. are set identifiers (cf. example 2.2). Therefore, the following definition

$u := \{ (\text{He}, \text{Harry}), (\text{She}, \text{Sally}), (\text{HisSons}, \text{Linus}), (\text{HisDaughters}, \text{Clio}),$
 $(\text{HerDaughters}, \text{Clio}), (\text{HerDaughters}, \text{Mary}) \}$

is a correct redex selecting relation for F in F' with $F \subseteq_{u, \Psi} F'$

since $u^*(F) \subseteq F'$ and thereby $F' \subseteq_u F$.

Furthermore, we can assume that the completing operator C generates formulas by means of which we are able to prove that

- Harry is not the target of a child-edge and, therefore, not the brother of another person,
- Harry is not the source of a wife-edge, and
- Gina is not the target of a wife-edge. ■

The relation u defined above is even maximal with respect to the number of object identifiers in F' which are bound to set identifiers in F . This is an important property of redex selecting relations, which are involved in the process of structure rewriting.

Having presented the definitions of structure schemas, schema consistent structures, and structure morphisms, we are now prepared to introduce *structure rewrite rules* as quadruples of *sets of closed formulas*. The application of structure rewrite rules will be defined as the construction of commuting diagrams in a similar way as it is done in the algebraic graph

grammar approach [8]. But note that we use redex selecting *relations* instead of (partial) mappings. Therefore, 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, which delete/copy/redirect arbitrarily large edge bundles.

Definition 3.8 Structure Rewrite Rule.

A quadruple $p := (AL, L, R, AR)$ with $AL, AR \in \mathcal{F}(\Sigma)$ and with $L, R \in \mathcal{L}(\Sigma)$ 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/embedding 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). ■

The following example defines a structure rewrite rule which is based on the structure selecting example 3.7. It marries two persons under certain preconditions, whereby each person adopts the other person's children.

Example 3.9 The Structure Rewrite Rule "Marry".

With F and Ψ defined as in example 3.7, the rewrite rule $Marry := (AL, L, R, AR)$ is defined as follows:

- (1) $AL := \Psi$.
- (2) $L := F$.
- (3) $R := F \cup \{ \text{edge}(\text{He}, \text{wife}, \text{She}), \text{edge}(\text{He}, \text{child}, \text{HerSons}), \text{edge}(\text{He}, \text{child}, \text{HerDaughters}), \text{edge}(\text{She}, \text{child}, \text{HisSons}), \text{edge}(\text{She}, \text{child}, \text{HisDaughters}) \}$.
- (4) $AR := \{ \}$. ■

Figure 1 displays the result of applying the structure rewrite rule *Marry* twice to the database of figure 1. Sally is now married to Harry and Fred to Clio. The identifier bindings of the first rewrite step are shown in example 3.7. In the second rewrite step, *He* is bound to Fred and *She* to Clio. Furthermore, all set identifiers of the rule's left- and right-hand side have empty bindings (Fred and Clio have no children). Both rewrite steps are executed as follows:

- Find suitable matches for the object identifiers of L (*She* and *He*) in the regarded database such that the corresponding preconditions in AL are satisfied.
- Extend the selected redex to all object set identifiers in L and bind each set identifier to a maximal set of objects in the database such that preconditions are still valid.
- Remove the image of L without the image of R from the database (empty in our case).
- Add an image of R without the image of L to the database (a *wife*-edge between the images of *He* and *She*).
- Test the postconditions AR . If any postcondition is violated, then abort all database modifications, return to step (1), and restart rule application with another redex.
- Finally, test whether the constructed database is schema consistent and treat inconsistency like violated postconditions.

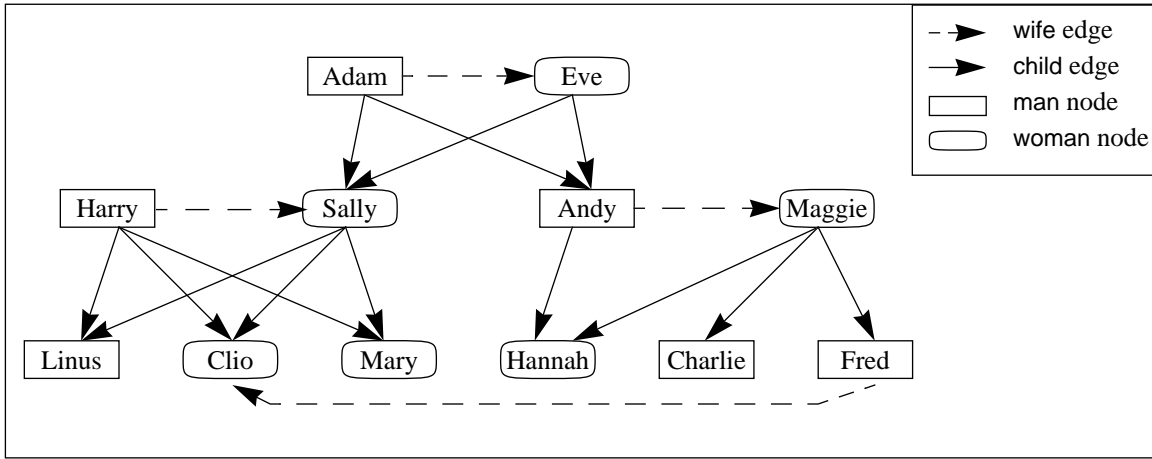


Fig. 3: A modified person database

Applications of the rewrite rule *Marry* do not have to worry about postconditions and schema inconsistencies. All necessary context conditions are already part of the preconditions *AL*. But we could also move the brother check from the set of preconditions *AL* to the set of postconditions *AR*. We could even drop the “already married” checks in *AL*, since the schema of example 2.11 contains integrity constraints preventing polygamy. Normally, integrity constraints are used instead of repeated postconditions, and postconditions for new objects in $R \setminus L$ replace very complex preconditions. It is still an open question whether any set of postconditions (referencing for instance derived properties of new objects) may be transformed into an equivalent set of preconditions.

Finally note that we have to take care about “dangling” edges in the general case of a person database rewrite rule, where L is not a subset of R . We have to guarantee that any formula $\text{node}(x, l)$ is removed together with all related $\text{edge}(x, e, y)$ and $\text{edge}(y', e', x)$ formulas. This may be accomplished by adding appropriate edge formulas with set identifiers (instead of y) to the rule’s left-hand side. In this way, we are able to overcome the problem of the algebraic double pushout approach with deletion of nodes with unknown context [8].

Definition 3.10 Schema Preserving Structure Rewriting.

$S := (\Phi, C) \in \mathcal{S}(\Sigma)$ and $F, F' \in \mathcal{L}(S)$. Furthermore, $p := (AL, L, R, AR) \in \mathcal{P}(\Sigma)$. The structure F' is **direct derivable** from F by applying p (write: $F \sim p \rightsquigarrow F'$) \Leftrightarrow

- (1) There is a morphism $u: L \rightsquigarrow F$ with: $L \subseteq_{u, AL} F$,
i.e. the via u selected redex in F respects the preconditions *AL*.
- (2) There is no morphism $\hat{u}: L \rightsquigarrow F$ with: $L \subseteq_{\hat{u}, AL} F$ and $u \subset \hat{u}$,
i.e. u selects a maximal substructure in F (with \subset being the inclusion for relations).
- (3) There is a morphism $w: R \rightsquigarrow F'$ with: $R \subseteq_{w, AR} F'$,
i.e. the via w selected subgraph in F' respects the postconditions *AR*.
- (4) The morphism w maps any new object identifier of R , which is not defined in L , onto a separate new object identifier in F' , which is not defined in F .
- (5) With $K := L \cap R$ the following property holds:
 $v := \{ (x, y) \in u \mid x \text{ is identifier in } K \} = \{ (x, y) \in w \mid x \text{ is identifier in } K \}$,
i.e. u and w are identical with respect to identifiers in the “gluing” structure K .
- (6) There exists a graph $H \in \mathcal{L}(\Sigma)$: $F \setminus (u^*(L) \setminus v^*(K)) = H = F' \setminus (w^*(R) \setminus v^*(K))$,
i.e. the intermediate result H is not required to be schema consistent. ■

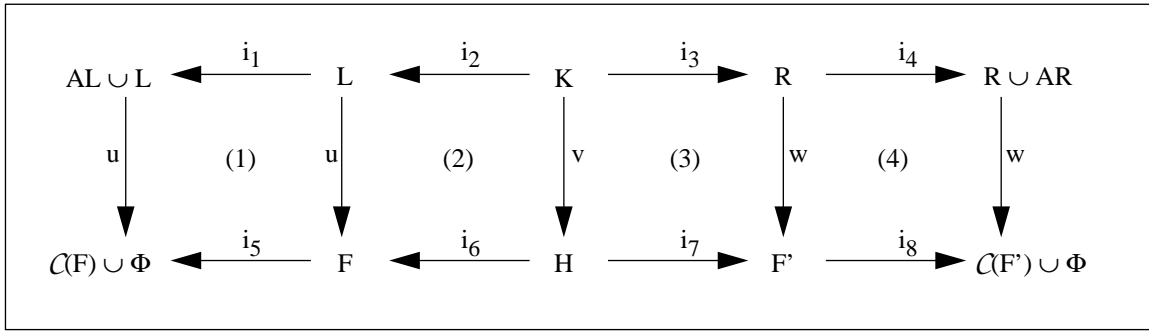


Fig. 4: Diagram for the application of a structure rewrite rule

Note that this definition of structure rewriting does not prohibit the selection of *homomorphic redices* with two identifiers o_1 and o_2 in L being mapped onto the same object in F , even if o_1 belongs to R and o_2 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 schema preserving structure rewriting above into an *effective procedure* for the application of a rule p to a schema consistent structure F . The execution of p proceeds as already explained before in definition 3.10 and is equivalent to the construction of the diagram in figure 4. Unfortunately, we can not guarantee that the process of computing derived data or checking pre- and postconditions as well as integrity constraints terminates in the general case. Therefore, we will introduce a special symbol “ ∞ ” for nonterminating computations in the next section, when we define the semantics of structure rewriting programs (transactions)..

Proposition 3.11 Structure Rewriting is Construction of Diagrams.

Assuming the terminology of definition 5.3 and $F \sim p \rightsquigarrow F'$ we are able to construct the diagram of fig. 6 (with i_1, \dots, i_8 being inclusions). This diagram has commuting subdiagrams (1) through (4).

Proof:

The existence of commuting subdiagrams (1) and (4) is guaranteed by definition 3.6. 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 :

We start with definition 3.10, condition (1):

$$\begin{array}{ll}
 u: L \rightsquigarrow F & \Leftrightarrow \text{def. 3.2, prop. 3.5, and } K \subseteq L & u^*(K) \subseteq u^*(L) \subseteq F \\
 & \Rightarrow \text{condition (5) of def. 3.10} & v^*(K) \subseteq u^*(L) \subseteq F \\
 & \Rightarrow \text{simple transformation} & v^*(K) \setminus (u^*(L) \setminus v^*(K)) \subseteq F \setminus (u^*(L) \setminus v^*(K)) \\
 & \Leftrightarrow \text{simple transformation} & v^*(K) \subseteq F \setminus (u^*(L) \setminus v^*(K)) \\
 & \Leftrightarrow \text{condition (6) of def. 3.10} & v^*(K) \subseteq H \\
 & \Leftrightarrow \text{def. 3.2 and prop. 3.5} & v: K \rightsquigarrow H.
 \end{array}$$

The rest of the proof follows directly from

$$i_2 \circ u = v \circ i_6 \quad \text{and} \quad v \circ i_7 = i_3 \circ w,$$

since v is a restriction of u or w onto the identifiers in K . ■

4. Recursively Defined Structure Rewriting Transactions

So far we would be able to define sets of schema 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] and are the fundamental concept of specification languages like PAGG [12] and PROGRES [37].

On the following pages we will develop a *denotational semantics* 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 schema consistent structure either succeeds and returns another schema consistent structure or fails without any structure modifying effects. Aspects of concurrency and isolation, normally associated with the term “transaction”, are ignored within this paper in accordance with its focus on sequential transformation processes. 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.

Figure 5 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 three binary BCF operators with

- def(A) being an action which succeeds for a given structure F, whenever A applied to F produces a defined result, and returns F itself,
- undef(A) being an action which is the complement of “def(A)” and succeeds for a given structure F, whenever A applied to F terminates with failure, and returns F itself,

```

<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> ")" |
                      "(" <BCF_Exp> "&" <BCF_Exp> ")" ;

```

Fig. 5: Syntax of transactions and BCF expressions

- $(A ; B)$ being an action which is the sequential composition of A and B, i.e. applies first A to a given structure F and then B to any “suitable” result of the application of B,
- $(A \parallel B)$ being an action which represents the nondeterministic choice between the application of A or B to a given structure F, and
- $(A \& B)$ being an action which applies both A and B to a given structure F and returns a common result of A and B.

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 F 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 $(A \parallel B)$ where A loops forever applied to a certain structure F 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 $(A \parallel B)$ 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.

The following example defines a transaction, which tries to complete a person database such that all persons are married afterwards. It uses the rewrite rule *Marry* of example 3.9 and assumes the existence of another rewrite rule *MarkUnmarried*. The important property of *MarkUnmarried* is that it may not be applied to a database which contains married persons only.

Example 4.1 A Person Database Rewriting Transaction.

$\text{Marry}^* = ((\text{def}(\text{MarkUnmarried}) ; \text{Marry}) ; \text{Marry}^*) \parallel \text{undef}(\text{MarkUnmarried})) . \blacksquare$

The transaction above initiates *Marry*-calls as long as the production *MarkUnmarried* is applicable. It terminates successfully, if and only if all persons are finally married. Otherwise, a database state will be reached, where neither the first nor the second branch of “ \parallel ” is executable. In this case, the transaction aborts without any database modifications. Note that the execution of *Marry*^{*} requires backtracking in the general case. Consider for instance the application of *Marry*^{*} to the database of figure 1. The transformation process may start with marrying Linus and *Marry* first and marrying Harry and Sally afterwards. The problem is now that no partner for Hannah is left over. Therefore, backtracking starts and another couple instead of Harry and Sally is married (e.g. Harry and Hannah).

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 [27]. In order to be able to deal with nonterminating computations we have to extend the semantic domain of binary relations over schema consistent structures:

Definition 4.2 Extended Semantic Domain.

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

$$\mathcal{D} := 2^{\mathcal{L}(S) \times (\mathcal{L}(S) \cup \{\infty\})}. \quad \blacksquare$$

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 F 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 prove that any chain in \mathcal{D} has a join, i.e. that any sequence of elements $(R^\alpha)_{\alpha \in \text{ordinal}}$ 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 4.3 Partial Order on Semantic Domain.

With $R, R' \in \mathcal{D}$ and S being the underlying structure schema, a suitable **partial order** “ \leq ” is defined as follows:

$$R \leq R' :\Leftrightarrow \forall F, F' \in \mathcal{L}(S): \quad (F, F') \in R \Rightarrow (F' = \infty \vee (F, F') \in R') \\ \wedge (F, \infty) \notin R \Rightarrow ((F, F') \in R \Leftrightarrow (F, F') \in R'). \quad \blacksquare$$

The relation R of the definition above *approximates* R' such that any input F 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 “ \leq ” is indeed a partial order but we have to prove its “chains have joins” condition:

Lemma 4.4 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:
 $R^\alpha \leq R^{\alpha+\beta}$.

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

$$R := \cup(R^\alpha) := \{ (F, F') \mid F' \neq \infty \wedge \exists \alpha: (F, F') \in R^\alpha \\ \vee F' = \infty \wedge \forall \alpha: (F, \infty) \in R^\alpha \}.$$

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:

$$\begin{aligned} \forall \alpha: R^\alpha \leq R : \\ (F, F') \in R^\alpha &\Rightarrow (F, F') \in R \vee F' = \infty. \\ (F, \infty) \notin R^\alpha &\Rightarrow \forall \beta \geq \alpha: (F, F') \in R^\alpha \Leftrightarrow (F, F') \in R^\beta \\ &\Rightarrow (F, F') \in R^\alpha \Leftrightarrow (F, F') \in R. \\ \forall R' \in \mathcal{D}: (\forall \alpha: R^\alpha \leq R') &\Rightarrow R \leq R' : \\ (F, F') \in R &\Rightarrow \exists R^\alpha: (F, F') \in R^\alpha \vee F' = \infty \\ &\Rightarrow (F, F') \in R' \vee F' = \infty. \\ (F, \infty) \notin R &\Rightarrow \exists R^\alpha: (F, \infty) \notin R^\alpha \\ &\Rightarrow \exists R^\alpha: (F, F') \in R \Leftrightarrow (F, F') \in R^\alpha \Leftrightarrow (F, F') \in R'. \quad \blacksquare \end{aligned}$$

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

Definition 4.5 BCF Expressions, Transactions, and their Semantics.

\mathcal{S} and \mathcal{D} are defined as in 4.3, and $\mathcal{P} \subseteq \mathcal{P}(\Sigma)$ is a set of structure rewrite rules. Then, $\mathcal{E}(\mathcal{P})$ denotes a set of **BCF expressions** and $\mathcal{T}(\mathcal{P})$ a set of **transactions**. Their context-free syntax is displayed in figure 5, and a **semantic function** $\mathcal{R}: \mathcal{E}(\mathcal{P}) \rightarrow \mathcal{D}$ is defined as follows⁷:

- (1) $(F, F') \in \mathcal{R}[\text{skip}] :\Leftrightarrow F = F' .$
- (2) $(F, F') \in \mathcal{R}[\text{loop}] :\Leftrightarrow F' = \infty .$
- (3) $(F, F') \in \mathcal{R}[p] :\Leftrightarrow \begin{aligned} &F \sim p \rightsquigarrow F', \quad \text{for any production } p \in \mathcal{P}(\Sigma) \\ &\vee F' = \infty, \quad \text{if execution of } p \text{ may not terminate}^8. \end{aligned}$
- (4) $(F, F') \in \mathcal{R}[\text{def}(A)] :\Leftrightarrow \begin{aligned} &\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[A] \wedge F = F' \\ &\vee (F, \infty) \in \mathcal{R}[A] \wedge F' = \infty . \end{aligned}$
- (5) $(F, F') \in \mathcal{R}[\text{undef}(A)] :\Leftrightarrow \begin{aligned} &(\neg \exists F'' : (F, F'') \in \mathcal{R}[A]) \wedge F = F' \\ &\vee (F, \infty) \in \mathcal{R}[A] \wedge F' = \infty . \end{aligned}$
- (6) $(F, F') \in \mathcal{R}[A ; B] :\Leftrightarrow \begin{aligned} &\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[A] \wedge (F'', F') \in \mathcal{R}[B] \\ &\vee (F, \infty) \in \mathcal{R}[A] \wedge F' = \infty . \end{aligned}$
- (7) $(F, F') \in \mathcal{R}[A \parallel B] :\Leftrightarrow (F, F') \in \mathcal{R}[A] \vee (F, F') \in \mathcal{R}[B] .$
- (8) $(F, F') \in \mathcal{R}[A \& B] :\Leftrightarrow \begin{aligned} &F' = \infty \wedge ((F, \infty) \in \mathcal{R}[A] \vee (F, \infty) \in \mathcal{R}[B]) \\ &\vee (F, F') \in \mathcal{R}[A] \wedge (F, F') \in \mathcal{R}[B] . \quad \blacksquare \end{aligned}$

The definitions above are straightforward with the exception of the treatment of the operators def and undef. The expressions def(A) and undef(A) loop forever if A loops forever. Furthermore, def(A) returns its input if A returns at least one defined result, and it terminates with failure if A terminates with failure. But undef(A) returns its input if and only if A fails, and it fails if and only if A has at least one defined result and may not loop forever. Therefore, undef is stricter than def with respect to the treatment of looping computations; the expression def(A) may return a defined result even if A may loop forever. From a practical point of view, this distinction may be justified as follows:

- Often, we would like to know whether A computes at least one defined result without evaluating all possible execution paths of A (after a successful path has been found).
- On the other hand, answering the question whether A fails is not possible without taking all execution paths of A into account (thereby running into any nonterminating execution branch of A).

Based on the definitions above we are now able to develop a fixpoint semantics for recursively defined transactions like *Marry** of example 4.1. For this purpose, we will use the following version of a fixpoint theorem (taken from [27]):

Proposition 4.6 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

$$f^\alpha = \left(\bigcup_{\beta < \alpha} f(f^\beta) \right) .$$

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

Proof: See appendix of [27]. ■

- 7) The definition of the semantics of (recursive) transactions themselves has to be postponed, until a suitable fixpoint theorem has been introduced and its preconditions are checked.
- 8) See remarks to def. 3.10 concerning eventually nonterminating evaluations of pre- and postconditions.

In order to be able to use theorem 4.6 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 $\mathcal{R}[t_1], \dots, \mathcal{R}[t_n]$ of t_1 to t_n the semantics of E is given by definition 4.5 and is denoted as follows:

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

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

$$\text{Marry}^* = ((\underline{\text{def}}(\text{MarkUnmarried}) ; \text{Marry}) \sqcup \underline{\text{undef}}(\text{MarkUnmarried})),$$

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

$$R = \mathcal{R}[((\underline{\text{def}}(\text{MarkUnmarried}) ; \text{Marry}) \sqcup \underline{\text{undef}}(\text{MarkUnmarried}))] [R]$$

holds. Theorem 4.6 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 4.7 BCF Operators are Monotonic Functionals.

With \mathcal{D} , $\mathcal{L}(S)$, and \mathcal{R} from definition 4.5, $P \subseteq \mathcal{P}(\Sigma)$, and $A, A', B \in \mathcal{E}(P)$, $\mathcal{R}[A] \leq \mathcal{R}[A']$ 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) $\mathcal{R}[A \sqcup B] = \mathcal{R}[B \sqcup A] \leq \mathcal{R}[B \sqcup A'] = \mathcal{R}[A' \sqcup B]$.
- (6) $\mathcal{R}[A \& B] = \mathcal{R}[B \& A] \leq \mathcal{R}[B \& A'] = \mathcal{R}[A' \& B]$.

Proof:

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

$F \in \mathcal{L}(S)$ with $(F, \infty) \notin \mathcal{R}[A]$

$$\Rightarrow (F, F') \in \mathcal{R}[A] \Leftrightarrow (F, F') \in \mathcal{R}[A']$$

$$\Rightarrow (F, F') \in \mathcal{R}[\underline{\text{def}}(A)] \Leftrightarrow (F, F') \in \mathcal{R}[\underline{\text{def}}(A')].$$

$F \in \mathcal{L}(S)$ with $(F, \infty) \in \mathcal{R}[A] \Rightarrow (F, \infty) \in \mathcal{R}[\underline{\text{def}}(A)]$

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

$$(F, F') \in \mathcal{R}[\underline{\text{def}}(A)]$$

$$\Rightarrow (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[A] \wedge F = F') \vee F' = \infty$$

$$\Rightarrow (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[A'] \wedge F = F') \vee F' = \infty$$

$$\Rightarrow (F, F') \in \mathcal{R}[\underline{\text{def}}(A')] \vee F' = \infty.$$

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

$F \in \mathcal{L}(S)$ with $(F, \infty) \notin \mathcal{R}[A]$

$$\Rightarrow (F, F') \in \mathcal{R}[A] \Leftrightarrow (F, F') \in \mathcal{R}[A']$$

$$\Rightarrow (F, F') \in \mathcal{R}[\underline{\text{undef}}(A)] \Leftrightarrow (F, F') \in \mathcal{R}[\underline{\text{undef}}(A')].$$

$F \in \mathcal{L}(S)$ with $(F, \infty) \in \mathcal{R}[A]$ implies:

$$(F, F') \in \mathcal{R}[\underline{\text{undef}}(A)] \Leftrightarrow F' = \infty$$

$$\Rightarrow (F, F') \in \mathcal{R}[\underline{\text{undef}}(A)] \Leftrightarrow (F, F') \in \mathcal{R}[\underline{\text{undef}}(A')] \vee F' = \infty.$$

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

$F \in \mathcal{L}(S)$ with $(F, \infty) \notin \mathcal{R}[A]$

$$\begin{aligned} &\Rightarrow (F, F') \in \mathcal{R}[A] \Leftrightarrow (F, F') \in \mathcal{R}[A'] \\ &\Rightarrow (F, F') \in \mathcal{R}[A ; B] \Leftrightarrow (F, F') \in \mathcal{R}[A' ; B]. \end{aligned}$$

$F \in \mathcal{L}(S)$ with $(F, \infty) \in \mathcal{R}[A] \Rightarrow (F, \infty) \in \mathcal{R}[A ; B]$

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

$$\begin{aligned} &(F, F') \in \mathcal{R}[A ; B] \\ &\Rightarrow (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[A] \wedge (F'', F') \in \mathcal{R}[B]) \vee F' = \infty \\ &\Rightarrow (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[A'] \wedge (F'', F') \in \mathcal{R}[B]) \vee F' = \infty \\ &\Rightarrow (F, F') \in \mathcal{R}[A' ; B] \vee F' = \infty. \end{aligned}$$

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

$F \in \mathcal{L}(S)$ with $\neg \exists F'' : ((F, F'') \in \mathcal{R}[B] \wedge (F'', \infty) \in \mathcal{R}[A])$

and with any $F'' \in \mathcal{L}(S)$:

$$\begin{aligned} &(F, F'') \in \mathcal{R}[B] \\ &\Rightarrow (F'', \infty) \notin \mathcal{R}[A] \\ &\Rightarrow (F'', F') \in \mathcal{R}[A] \Leftrightarrow (F'', F') \in \mathcal{R}[A'] \\ &\Rightarrow (F, F') \in \mathcal{R}[B ; A] \Leftrightarrow (F, F') \in \mathcal{R}[B ; A']. \end{aligned}$$

$F \in \mathcal{L}(S)$ with $\exists F'' : ((F, F'') \in \mathcal{R}[B] \wedge (F'', \infty) \in \mathcal{R}[A])$

$$\Rightarrow (F, \infty) \in \mathcal{R}[B ; A]$$

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

$$\begin{aligned} &(F, F') \in \mathcal{R}[B ; A] \\ &\Rightarrow (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[B] \wedge (F'', F') \in \mathcal{R}[A]) \vee F' = \infty \\ &\Rightarrow (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}[B] \wedge (F'', F') \in \mathcal{R}[A']) \vee F' = \infty \\ &\Rightarrow \mathcal{R}[B ; A'] \vee F' = \infty. \end{aligned}$$

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

$F \in \mathcal{L}(S)$ with $(F, \infty) \notin \mathcal{R}[A]$

$$\begin{aligned} &\Rightarrow (F, F') \in \mathcal{R}[A] \Leftrightarrow (F, F') \in \mathcal{R}[A'] \\ &\Rightarrow (F, F') \in \mathcal{R}[A \parallel B] \Leftrightarrow (F, F') \in \mathcal{R}[A' \parallel B]. \end{aligned}$$

$F \in \mathcal{L}(S)$ with $(F, \infty) \in \mathcal{R}[A] \Rightarrow (F, \infty) \in \mathcal{R}[A \parallel B]$

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

$$\begin{aligned} &(F, F') \in \mathcal{R}[A \parallel B] \\ &\Rightarrow (F, F') \in \mathcal{R}[A] \vee (F, F') \in \mathcal{R}[B] \\ &\Rightarrow (F, F') \in \mathcal{R}[A'] \vee (F, F') \in \mathcal{R}[B] \vee F' = \infty \\ &\Rightarrow (F, F') \in \mathcal{R}[A' \parallel B] \vee F' = \infty. \end{aligned}$$

ad (6): $\mathcal{R}[A \& B] \leq \mathcal{R}[A' \& B]$:

$F \in \mathcal{L}(S)$ with $(F, \infty) \notin \mathcal{R}[A]$

$$\begin{aligned} &\Rightarrow (F, F') \in \mathcal{R}[A] \Leftrightarrow (F, F') \in \mathcal{R}[A'] \\ &\Rightarrow (F, F') \in \mathcal{R}[A \& B] \Leftrightarrow (F, F') \in \mathcal{R}[A' \& B]. \end{aligned}$$

$F \in \mathcal{L}(S)$ with $(F, \infty) \in \mathcal{R}[A] \Rightarrow (F, \infty) \in \mathcal{R}[A \& B]$

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

$$\begin{aligned} &(F, F') \in \mathcal{R}[A \& B] \\ &\Rightarrow (F, F') \in \mathcal{R}[A] \wedge (F, F') \in \mathcal{R}[B] \vee F' = \infty \\ &\Rightarrow (F, F') \in \mathcal{R}[A'] \wedge (F, F') \in \mathcal{R}[B] \vee F' = \infty \\ &\Rightarrow (F, F') \in \mathcal{R}[A' \& B] \vee F' = \infty. \end{aligned}$$



Corollary 4.8 Fixpoint Semantics for Transactions.

Let t_1 to $t_n \in \mathcal{T}(P)$ be transactions which contain in their bodies E_1 to $E_n \in \mathcal{E}(P)$ at most calls to transactions t_1 through t_n and calls to rewrite rules of $P \subseteq \mathcal{P}(\Sigma)$, i.e.

$$t_i = E_i[t_1, \dots, t_n], \dots, t_n = E_n[t_1, \dots, t_n].$$

With $\mathcal{L}(S)$ being the corresponding class of schema consistent structures, the following propositions hold:

- (1) t_1, \dots, t_n have unique least fixpoints $\mathcal{R}[t_1], \dots, \mathcal{R}[t_n] \in \mathcal{D}$.
- (2) Approximations of these fixpoints may be constructed in the following way:
 $R_1^0 := \mathcal{L}(S) \times \{\infty\}; \dots; R_n^0 := \mathcal{L}(S) \times \{\infty\}$,
 $R_1^{k+1} := \mathcal{R}[E_1][R_1^k, \dots, R_n^k]; \dots; R_n^{k+1} := \mathcal{R}[E_n][R_1^k, \dots, R_n^k]$,
 where $\mathcal{R}[E_i]: \mathcal{D}^n \rightarrow \mathcal{D}$ takes approximations for t_1, \dots, t_n and yields a new approximation for t_i by applying definition 4.5 to expression E_i .

Proof:

ad (1): Follows directly from theorem 4.6 and lemmata 4.4 and 4.7.

ad (2): The relations R_i^k are indeed approximations for t_1, \dots, t_n such that for any i, k :
 $R_i^0 \leq \dots \leq R_i^k \leq \mathcal{R}[t_i]$. This follows directly from theorem 4.6 and lemmata 4.4 and 4.7 with

$$\forall i, l: R_i^l = \cup_{k \leq l} R_i^k \leq \mathcal{R}[t_i] \text{ and } \cup_{k \in \mathbb{N}} R_i^k = \mathcal{L}(S) \times \{\infty\}. \quad \blacksquare$$

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.

Equipped with a fixpoint computing function \mathcal{R} we are now able to define programmed structure rewriting systems and their generated structure languages.

Definition 4.9 Programmed Structure Rewriting Systems.

Assuming the vocabulary of definition 4.5, a quadruple $\text{PSRS} := (A, P, T, t)$ is a **programmed structure rewriting system** with respect to a structure schema $S \in \mathcal{S}(\Sigma)$ iff:

- (1) $A \in \mathcal{L}(S)$ is the schema consistent initial structure.
- (2) $P \subseteq \mathcal{P}(\Sigma)$ is a set of structure rewrite rules.
- (3) $T \subseteq \mathcal{T}(P)$ is a set of (recursively) defined transactions over P .
- (4) $t \in T$ is the “main transaction” of PSRS. \blacksquare

Definition 4.10 Language of a Programmed Structure Rewriting System.

With $\text{PSRS} := (A, P, T, t)$ as in def. 4.9, the **language** $\mathcal{L}(\text{PSRS})$ is defined as follows:

$$F \in (\text{PSRS}) :\Leftrightarrow (A, F) \in \mathcal{R}(t). \quad \blacksquare$$

The definition above states that the language of a programmed structure rewriting system consists of all those structures which may be generated by applying a main transaction t to the initial structure A . The transaction t calls structure rewrite rules from P and other transactions from T , which in turn contain calls of rewrite rules and transactions. A distinction between terminal and nonterminal structures has not been made in the preceding sections. Hence, any result of t is per definition a (terminal) element of the generated language.

5. Related Work

The purpose of this section is to compare our rewriting approach with a number of related approaches, which are the nonparallel versions of

- the algebraic double pushout graph grammar approach in [8],
- its generalization in the form of so-called structure grammars in [7],
- the new algebraic single pushout graph grammar approach as presented in [18],
- the expression oriented algorithmic graph grammar approach of [23],
- the structured graph grammar approach of [15],
- the programmed graph grammar approach of [12],
- and finally the “programmed derivation of relational structures” approach in [22].

Before going into details, we have to clarify the meaning of the statement “that an approach A is *more expressive* than another approach B”. From a theoretical point of view, it means that we are able to define a language \mathcal{L} with approach A, whenever we are able to define this language with approach B. Since different approaches support different data (graph) models we have to require more precisely: any grammar (rewriting system) of approach B, which defines a language \mathcal{L}_B , may be replaced by a grammar of approach A, which defines a language \mathcal{L}_A , and a bijection between \mathcal{L}_A and \mathcal{L}_B . From a practical point of view, the requirement above is too weak, since it includes even cases where an exponentially greater number of productions (rules) of approach A is necessary to replace a number of productions of approach B. Therefore, we say henceforth

Definition 5.1 Expressiveness of Rewriting Approaches.

A rewriting approach A is at least as **expressive** as another approach B if and only if:

Any production or program p_B of approach B may be translated into an **equivalent** production or program p_A of approach A such that:

p_A derives H_A from $G_A \Leftrightarrow p_B$ derives H_B from G_B and $f(G_A) = G_B$ and $f(H_A) = H_B$, with f being a bijection between the generated languages \mathcal{L}_A and \mathcal{L}_B . ■

Based on this definition we are able to discuss to which extent programmed structure rewriting systems are *more expressive* than the above listed approaches. Primarily, none of them allows for the construction of schemas, which are a prerequisite for the definition of global integrity constraints and derived properties⁹. Furthermore, application conditions are either not supported or considerably less expressive than pre- and postconditions of structure rewrite rules. Therefore, we will neglect these topics in the sequel, and focus on the discussion of rewriting programs and rewriting rules without application conditions. Due to lack of space, this discussion has to be on an informal level, and the more or less trivial problem of describing data models of competing approaches as scheme consistent structures had to be omitted.

Any production of the *algebraic double pushout approach* [8] may be translated into an equivalent structure rewrite rule as long as the morphisms from its explicitly defined gluing graph K to its left-hand side L and its right-hand R are monomorphisms (which is usually required). In this case, the gluing graph K may be omitted and implicitly defined as the intersection of L and R. Hence, the translation process is straightforward with the exception of the so-called “dangling edge” and “identification” conditions. These conditions must be translat-

9) Although [7] and especially [17] with their combination of heterogeneous algebras and graphs are a significant step into this direction for the algebraic double and single pushout approach.

ed into preconditions of structure rewrite rules, which prevent their application in the case of afterwards dangling edges (cf. remarks to example 3.9) and preservation/deletion conflicts (cf. remarks to definition 3.10). The *single pushout approach* [18] differs from the double pushout approach (from a user’s point of view) in as much as its productions are able to delete dangling edges. This behavior may be simulated in our approach with the help of object set identifiers (cf. remarks to example 3.9).

The main “behavioral” differences between productions of the single pushout approach and the *expression oriented algorithmic approach* in [23] are that productions of the algorithmic approach (1) always delete the complete image of their left-hand side and (2) have very complex embedding rules for redirecting and copying context edges of the deleted image. Remember that the main motivation for the introduction of object set identifiers over here was the treatment of complex embedding rules. Therefore, any production of the expression oriented approach may be translated into a single structure rewrite rule.

The next approach listed above are the *structured graph grammars* of [15]. They combine the concept of a common subgraph of a production’s left- and right-hand side with the concept of embedding rules and offer different options for the treatment of dangling edges. To translate productions of this approach into structure rewrite rules, we have merely to combine the translation techniques for the algebraic double pushout and expression oriented approach.

Programmed graph grammars [12] are another approach which combines the concept of a common subgraph of a production’s left- and right-hand side with complex embedding rules. These embedding rules are defined in a purely graphical manner and may consist of arbitrarily complex subgraph patterns. Such a subgraph pattern can be translated into an occurrence of an object set identifier with additional restrictions as long as the following condition is not violated: the subgraph pattern may not contain two different nodes which are both connected to the production’s left-hand side¹⁰. Furthermore, three operators are offered for programming with productions (supporting concatenation, branching, and iteration). The translation of these operators into BCF expressions is possible but complicated by the fact that a sequence of production calls of [12] is successful as long as one of the listed productions may be applied (all others are skipped). In this way, the problem of backtracking out of dead-ends can be avoided but concatenation has a rather counterintuitive semantics (for further details see discussion of related work in [35]).

Finally, we have to discuss the “*programmed derivation of relational structures*” approach in [22]. Its rewrite rules are equivalent to our rewrite rules without set identifiers and pre- or postconditions. On the programming level we have the following differences: [22] has its main focus on parallel programming (not supported over here), whereas BCF expressions offer the additional operator $\&$ (for intersection of subprogram outputs) as well as def and undef (for testing whether the output of a subprogram is empty or not). Parallel composition of subprograms as well as testing their emptiness are both very valuable means and require completely different formal treatments. Parallel composition, on one hand, with its “interleaving semantics” excludes the definition of a program’s semantics as a function of the semantics of its subprograms. Testing success or failure of subprograms, on the other hand, enforces the introduction of a special symbol “ ∞ ” for nonterminating computations and the usage of a rather complicated partial order for the resulting semantic domain. Further investigations are necessary to check whether a combination of both approaches is possible or not.

10)In this case, bindings of object set identifiers to sets of object identifiers are no longer independent from each other, which is a prerequisite of the definition of redex selecting relations over here.

6. 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. 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. [11, 31]), 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, 14, 29, 39]).

Last but not least the new approach provides us with a formal definition of partial, nondeterministic, and even recursive *structure rewriting programs*, termed transactions. We added a new symbol “ ∞ ” to the domain of structures, which represents unknown results or nonterminating computations. Thereby, we were able to overcome the difficulties with operators, which test success or failure of subprograms, in the presence of recursion (cf. [35] for a more detailed discussion about these problems). These operators are a prerequisite for writing programs like “try first to execute subprogram A and, if and only if A fails, try to execute B”.

Note that [27], 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, since structure rewrite rules play here about the same role as simple assignments in [27]. The definition of their intended semantics as binary relations is given in definition 3.10, but it is a yet unsolved problem how to “push” postconditions through rewrite rules for obtaining their corresponding weakest preconditions (see question (3) below).

Beside the problem to extend the presented new approach such that *parallel structure rewriting* is supported, the following questions should be studied in the future:

- (1) Which restrictions are sufficient to guarantee that subdiagrams (2) and (3) of figure 6 are pushouts beyond those subcases identified in [7]?
- (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 schema 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 envi-*

ronment. 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 compiler backends for Modula-2 and C. A first release of these tools and the underlying graph-oriented database system GRAS [16] 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, Springer Verlag (1986)
- [2] Bry F., Manthey R., Martens B.: *Integrity Verification in Knowledge Bases*, in: [38], 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, Springer Verlag (1979)
- [5] Courcelle B.: *Graphs as Relational Structures: An Algebraic and Logical Approach*, in: [9], 238-252
- [6] Dassow J., Paun G.: *Regulated Rewriting in Formal Language Theory*, EATCS 18, Springer Verlag (1989)
- [7] Ehrig H., Habel A., Kreowski H.-J., Parisi-Presicce F.: *From Graph Grammars to High-Level Replacement Systems*, in: [9], 269-291
- [8] Ehrig H.: *Introduction to the Algebraic Theory of Graph Grammars (a Survey)*, in: [4], 1-69
- [9] Ehrig H., Kreowski H.-J., Rozenberg G.: *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Springer Verlag (1991)
- [10] 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, acm Press (1992), 135-167
- [11] Gallaire H.: *Logic and Data Bases*, New York: Plenum Press (1978)
- [12] Göttler H.: *Graphgrammatiken in der Softwaretechnik*, IFB 178, Springer Verlag (1988)
- [13] 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, Fachgruppe Informatik, RWTH Aachen, Germany (1993)
- [14] Van Hentenryck P.: *Constraint Satisfaction in Logic Programming*, MIT Press (1989)
- [15] Kreowski H.-J., Rozenberg G.: *On Structured Graph Grammars: Part I and II*, Technical Report 3/88, University of Bremen, FB Mathematik/Informatik (1988)
- [16] Kiesel N., Schürr A., Westfechtel B.: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*, in: CASE '93 Proc. 6th Int Workshop on Computer-Aided Software Engineering, IEEE Computer Society Press (1993), 272-286

- [17] Korff M.: *Algebraic Transformation of Equationally Defined Graph Structures*, TR No. 92/32, Fachbereich Informatik, TU Berlin, Germany (1992)
- [18] Löwe M.: *Algebraic Approach to Graph Transformation Based on Single Pushout Derivations*, TR No. 90/5, Fachbereich Informatik, 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*, 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, acm Press (1990), 325-362
- [21] Minker J.: *Perspectives in Deductive Databases*, in: The Journal of Logic Programming, Elsevier Science Publ. (1988), 33-60
- [22] Maggiolo-Schettini A., Winkowski J.: *Programmed Derivations of Relational Structures*, in: [9], 582-598
- [23] Nagl M.: *A Tutorial and Bibliographical Survey on Graph Grammars*, in: [4], 70-126
- [24] Nagl M.: *Graph Technology Applied to a Software Project*, in: [30], 303-322
- [25] Naqvi Sh.A.: *Some Extensions to the Closed World Assumption in Databases*, in: [1], 341-348
- [26] Naqvi Sh.A.: *Negation as Failure for First-Order Queries*, in: Proc. 5th SIGACT-SIGMOD Symp. on Principles of Database Systems, acm Press (1986), 114-122
- [27] Nelson G.: *A Generalization of Dijkstra's Calculus*, in ACM Transactions on Programming Languages and Systems, vol. 11, no. 4, acm Press, USA (1989), 517-561
- [28] Nagl M., Schürr A.: *A Specification Environment for Graph Grammars*, in: [9], 599-609
- [29] Naqvi Sh.A., Tsur Sh.: *Data and Knowledge Bases*, IEEE Computer Society Press (1989)
- [30] Rozenberg G., Salomaa A.: *The Book of L*, Springer Verlag (1986)
- [31] Rybinski H.: *On First-Order-Logic Databases*, in: ACM Transaction on Database Systems, Vol. 12, No.3, acm Press (1987), 325-349
- [32] Schmidt G., Berghammer R.: *Proc. Int. Workshop on Graph- Theoretic Concepts in Computer Science (WG '91)*, LNCS 570, Springer Verlag(1991)
- [33] Schneider H.J., Ehrig H. (eds.): *Proc. Int. Workshiop on Graph Transformations in Computer Science*, LNCS 776, Springer Verlag (1994)
- [34] Schürr A.: *PROGRES: A VHL-Language Based on Graph Grammars*, in: [9], 641-659
- [35] Schürr A.: *Operationales Spezifizieren mit Programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungen und Werkzeuge*, Deutscher Universitätsverlag (1991)
- [36] Schürr A.: *Logic Based Structure Rewriting Systems*, in: [33], 341-357
- [37] Schürr A., Zündorf A.: *Non-Deterministic Control Structures for Graph Rewriting Systems*, in: [32], 48-62
- [38] Voronkov A. (ed.): *Logic Programming*, LNCS 592, Springer Verlag (1991)
- [39] Thaise A. (ed.): *From Standard Logic to Logic Programming*, John Wiley & Sons Ltd. (1989)