

Programmed Graph Replacement Systems

Andy Schürr

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

Various forms of programmed graph replacement systems as extensions of context-sensitive graph replacement systems have been proposed until today. They differ considerably with respect to their underlying graph models, the supported forms of graph replacement rules, and offered rule regulation mechanisms. Some of them have additional constructs for the definition of graph schemata, derived graph properties, and so forth. It is rather difficult to develop precise and compact descriptions of programmed graph replacement systems, a necessary prerequisite for any attempt to compare their properties in detail. Programmed Logic-based Structure Replacement (PLSR) systems are a kind of intermediate definition language for this purpose. They treat specific graph classes as sets of predicate logic formulas with certain properties, so-called structures. Their rules preserve the consistency of manipulated structures and use nonmonotonic reasoning for checking needed pre- and postconditions. So-called Basic Control Flow (BCF) expressions together with an underlying fixpoint theory provide needed means for programming with rules. This chapter introduces first the basic framework of PLSR systems and studies afterwards the essential properties of context-sensitive graph replacement approaches themselves as well as popular rule regulation mechanisms.

1 Introduction

The history of graph replacement systems starts 25 years ago with two seminal papers about so-called “web grammars” [53] and “Chomsky systems for partial orders” [60]. From the beginning graph replacement systems have been used and developed to solve “real world” problems in the fields of computer science itself, biology, etc. [1,50]. One of the first surveys about graph replacement systems [43] (and its short version in [44]) distinguishes three different types of applications:

- (1) Describing or *generating* known *languages of graphs* which have for instance certain graph-theoretical properties or model growing plants.
- (2) *Recognizing languages of graphs* which model for instance the underlying logical structure of natural language sentences or scanned images and office documents.
- (3) *Specifying new classes of graphs* and graph transformations which represent for instance databases and database manipulations or software documents and document manipulating tools.

For solving problems in the first category pure graph replacement systems are often sufficient and accompanying tools are not urgently needed. The second category of problems requires classes of graph replacement systems which are powerful enough to describe interesting languages of graphs, but which are restricted enough so that efficiently working parsing or execution tools can be realized. The design of adequate classes of graph replacement systems is still subject of ongoing research activities and outside the scope of this paper (cf. Section 5 of Chapter 2 in this volume as well as [22] and [56]).

The third category of problems needs graph replacement systems as a kind of executable specification or even *very high level programming language*. In order to be able to fulfil the resulting requirements, many new concepts had to be added to “pure” first generation graph replacement systems (see [19] and especially [6] for a discussion of needed extensions). One of the most important extensions was the introduction of new means for controlling the application of graph replacement rules. This led to the definition of programmed graph replacement systems, which are the main topic of this contribution.

1.1 Programmed Graph Replacement Systems in Practice

For historical reasons mainly, almost all earlier proposals for programmed graph replacement systems [9,10,21,26,36,43,62] belong to one of the main branches of graph replacement systems, the *algorithmic, set-theoretic approach* (cf. Chapter 2 of this volume), and not to the *algebraic, category-theory approach* (see Chapter 4 and 5 of this volume). Until now, two complete implementations of programmed graph replacement systems have been built. The first one, **Programmed Attributed Graph Grammars** (PAGG), was mainly used for specifying graph editors and CAD systems [24,25,26]. The second one, **PROgrammed Graph REplacement Systems** (PROGRES), is tightly coupled to the development of software engineering tools [51,63,64,74]. PROGRES and its predecessors were and are used within the project IPSEN for specifying internal data structures and tools of **I**ntegrated **P**roject **S**upport **E**nvironments [19,20,45,46]. At the beginning, pure algorithmic graph replacement systems were used [43]. But soon it became evident that additional means for defining complex rule application conditions, manipulating node attributes etc. are needed. Appropriate extensions were suggested in [21] and later on in [36]. But even then there was no support for

- separating the definition of static graph *integrity constraints* from dynamic graph manipulating operations as database languages separate the definition of database schemata from the definition of database queries or manipulations,

- specifying *derived attributes* and *relations* in a purely declarative manner in a similar style as attribute tree grammars or relational languages deal with derived data,
- and solving typical generate and test problems by using implicitly available means for *depth-first search* and *backtracking* in the same manner as Prolog deals with these problems.

Therefore, various extensions of graph replacement systems were studied which resulted finally in the development of the programming language PROGRES and its integrated programming support environment.^a

1.2 Programmed Graph Replacement Systems in Theory

To produce a formal definition of programmed graph replacement systems like PAGG or PROGRES by means of set theory only is a very difficult and time-consuming task. Especially the definition of control structures, derived graph properties, and graph schemata constitutes a major problem. And switching from set theory to the formalism of category theory does not solve it at all. Both the category-based **Double Push Out** (DPO) approach (cf. Chapter 5 of this volume) and the **Single Push Out** approach (cf. Chapter 4 of this volume) have about the same difficulties with the definition of integrity constraints, derived information, and the like.

Therefore, it was necessary to establish another *framework for the formal definition of these application-oriented graph replacement systems*. The following observations had a major influence on the development of this framework:

- *Relational structures* are an obvious generalization of various forms of graphs or hypergraphs [42].
- *Logic formulas* are well-suited for defining required properties of generated graph languages (cf. Chapter 1 and 2 of this volume)
- *Nonmonotonic reasoning* is appropriate for inferring derived data and verifying integrity constraints in deductive database systems [41,52].
- *Fixpoint theory* is the natural candidate for defining the semantics of partially defined, nondeterministic, and recursive graph replacement programs [39,49].

Combining these sources of inspiration, first logic-based graph replacement systems were developed and used to produce a complete definition of the language PROGRES [64]. Later on they were generalized to **Logic Based Structure Replacement** (LBSR) systems [65] and **Programmed Logic-based Structure Replacement** (PLSR) systems [66].

^asee <http://www-i3.informatik.rwth-aachen.de/research/progres/index.html>

1.3 Contents of the Contribution

The presentation of LBSR and PLSR systems over here as well as a survey of *application-oriented* and/or *programmed context-sensitive* graph replacement systems is organized as follows:

- Section 2 introduces LBSR systems and demonstrates how they may be used to specify graph schemata, schema consistent graphs as well as consistency preserving graph transformations.
- Section 3 presents a basic set of control structures for programming with graph replacement rules together with a denotational semantics definition-based on fixpoint theory. Using these control structures, LBSR systems are extended to PLSR systems.
- Section 4 compares typical representatives of application-oriented algorithmic graph replacement systems and sketches their translation into LBSR systems. Furthermore it discusses their relationships to the DPO and SPO families of algebraic approaches.
- Section 5 finally surveys a collection of currently popular rule regulation mechanisms and translates them into basic control structures of PLSR systems.

Other topics, like the design of a readable syntax and a static type system for the language PROGRES or the implementation of tools which support programming with graph replacement systems, are outside the scope of this volume [63,64,66].

2 Logic-Based Structure Replacement Systems

Nowadays, a surprisingly large variety of different graph replacement formalisms is existent. Some of them have a rather restricted expressiveness and are well-suited for proving properties of generated graph languages. Some of them are very complex and mainly used for specification and rapid prototyping purposes. Even their underlying graph data models differ to a great extent with respect to the treatment of edges, attributes, etc.

In order to be able to reason about differences or common properties of these graph replacement approaches and their underlying data models in Section 4, a general framework is urgently needed. Such a framework has to provide proper means for the definition of specific graph models and graph replacement approaches. One attempt into this direction, so-called **H**igh **L**evel **R**eplacement (HLR) systems, is based on the construction of categories with certain properties [16]. It generalizes algebraic graph replacement approaches and incorporates algebraic specification technologies for the definition of at-

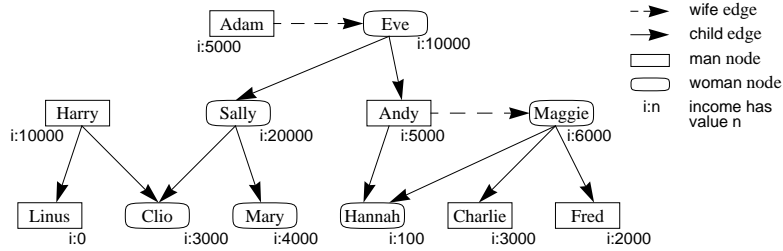


Figure 1: The running example, a person database

tribute domains (see also [32,37]). Until now, HLR systems still have problems with formalizing derived graph properties or graph integrity constraints as they are used within the graph grammar specification language PROGRES.

Therefore, another *generic framework* was developed which comprises many different graph replacement systems as special cases and which is the underlying formalism of the specification language PROGRES. It combines elements from algorithmic and algebraic graph replacement systems with elements from deductive database systems. The outcome of the cross-fertilization process, **Logic-Based Structure Replacement (LBSR)** systems, are the topic of this section.

The formalism of LBSR systems is not intended to be directly used as a specification language, but provides the fundament for the definition

- of specific graph models as special cases of relational structures
- and of a whole family of accompanying graph replacement notations.

The presentation of LBSR systems over here will be accompanied by a number of examples, which sketch how human-readable graph grammar specifications written in a PROGRES like notation may be translated into them. By means of these examples the reader will hopefully understand how LBSR systems may be used to define the semantics and compare the intended behavior of specific graph replacement systems.

The rest of this section is organized as follows:

- Subsection 2.1 repeats some *basic terminology* of predicate logic and contains the formal definition of (relational) structures and structure schemata as sets of formulas.
- Subsection 2.2 deals with *substructure selection*, i.e. finding matches for left-hand sides of rules with additional application conditions.
- Subsection 2.3 introduces then the *structure replacement formalism* itself.
- Subsection 2.4 finally *summarizes* the main properties of LBSR systems and their relationships to approaches like HLR systems.

Within all these subsections 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 and includes the definition of derived relationships, integrity constraints, and complex update operations (Figure 1 displays a sample database).

2.1 Structure Schemata and Schema Consistent Structures

This subsection introduces basic terminology of predicate logic. Furthermore, it explains modeling 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 modeled as a directed graph. In this way, we are able to demonstrate that graphs are a special case of structures and that graph replacement is a special case of structure replacement. Nevertheless, all definitions and propositions of the following sections are independent of the selected graph model and its encoding.

Definition 2.1 Signature

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

- (1) \mathcal{A}_F is an alphabet of function symbols^b (including constants).
- (2) \mathcal{A}_P is an alphabet of predicate symbols.^c
- (3) \mathcal{V} is a special alphabet of object identifier constants.
- (4) \mathcal{W} is a special alphabet of constants representing sets of objects.
- (5) \mathcal{X} is an alphabet of logical variables used for quantification purposes. \square

The signature definition above introduces names for specific objects and sets of objects as special constants. These constants will play an important role for the selection of LBSR rule matches. Elements of \mathcal{X} in a rule's left-hand side denote and match single objects of a given structure, whereas elements of \mathcal{W} denote and match sets of objects of a given structure. Please note that \mathcal{W} is *not* an alphabet of logical set variables, i.e. we do not use a monadic second order logic as in Chapter 1 of this volume. Quantification over set variables is needed for the definition of many graph-theoretic properties, but not a strict necessity for the formal definition of graph replacement formalisms studied over here. Hence we will use a first order logic which makes no distinction between constant symbols in \mathcal{A}_F (functions of arity 0), \mathcal{V} , and \mathcal{W} . Nevertheless, it is necessary to distinguish these three different kinds of constants within Subsection 2.2 and 2.3, where they play very different roles for the application of structure replacement rules.

^bA family of alphabets of symbols if we take domain and range of functions into account.

^cA family of alphabets of symbols if we take arities of predicates into account.

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 above 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. Furthermore, each person has an integer-valued **income** attribute.

Example 2.2 Signature for a Person Database.

The graph signature for fig. 1 is $\Sigma := (\mathcal{A}_F, \mathcal{A}_P, \mathcal{V}, \mathcal{W}, \mathcal{X})$ with:

- (1) $\mathcal{A}_F := \{ \text{child, woman, wife, man, person, income, integer, 0, 1, \dots, +} \}$.
- (2) $\mathcal{A}_P := \{ \text{node, edge, attr, type, \dots} \}$.
- (3) $\mathcal{V} := \{ \text{He, She, Value, \dots} \}$.
- (4) $\mathcal{W} := \{ \text{HerChildren, HisChildren, \dots} \}$.
- (5) $\mathcal{X} := \{ \text{x1, x2, \dots} \}$. □

The alphabets \mathcal{V} , \mathcal{W} , and \mathcal{X} above are sets of arbitrarily chosen constant or variable names. The set \mathcal{A}_F , on the other hand, contains just all needed symbols for node labels, edge labels, attribute types, attribute values, and evaluation functions.

The alphabet \mathcal{A}_P , finally contains four predicate symbols which are very important for the more or less arbitrarily selected encoding of directed, attributed graphs, where each node has a distinct class as its label and where any attribute value has a designated type. They have the following interpretation:

- $\text{node}(x, l)$: graph contains a node x with label l ,
- $\text{edge}(x, e, y)$: graph contains edge with label e from source x to target y ,
- $\text{attr}(x, a, v)$: attribute a at node x has value v ,
- $\text{type}(v, t)$: attribute value v has type t .

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

Definition 2.3 Σ -Term und Σ -Atom.

$\mathcal{T}_{\mathcal{X}}(\Sigma)$ is the set of all Σ -**terms** (in the usual sense) which contain function symbols and constants of \mathcal{A}_F , free variables of \mathcal{X} , and additional identifier symbols of \mathcal{V} and \mathcal{W} . $\mathcal{A}(\Sigma)$ is the set of all **atomic formulas** (Σ -**atoms**) over $\mathcal{T}_{\mathcal{X}}(\Sigma)$ which contain predicate symbols of \mathcal{A}_P and “=” for expressing the equality of two Σ -terms. Finally $\mathcal{T}(\Sigma) \subset \mathcal{T}_{\mathcal{X}}(\Sigma)$ is the set of all those terms which do not contain any symbols of \mathcal{X} , \mathcal{V} , and \mathcal{W} , i.e. do not have variables or object (set) identifiers as their leaves. □

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

$\mathcal{F}(\Sigma)$ is the set of all sets of closed^d first order predicate logic formulas (Σ -**formulas**) which have $\mathcal{A}(\Sigma)$ 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. \square

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

Definition 2.5 Σ -Structure.

A set of closed 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 “ $\tau_1 = \tau_2$ ”. \square

Example 2.6 A Person Database Structure.

The following structure is a set of formulas which has the graph F of figure 1 as a model:

$$F := \{ \text{node(Adam, man), node(Eve, woman), node(Sally, woman), } \dots, \\ \text{attr(Adam, income, 5000), attr(Eve, income, 10000), } \dots, \\ \text{edge(Adam, wife, Eve), edge(Eve, child, Sally), } \dots \}. \quad \square$$

The set of formulas F has many *different* graphs as *models*. One of those models is the graph of Figure 1, but there are many others in which we are not interested in. Some of them may contain additional nodes and edges, and some use different identifiers like **Eve** and **Sally** for the same graph node. 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 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 tackled by a number of quite different

^dRequiring 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.

approaches either based on the so-called *closed world assumption* or by using *nonmonotonic reasoning* capabilities (cf. [40,47,48]). 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 chapter 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 first-order predicate logic only.

Definition 2.7 Σ -Structure Completing Operator.

A function $\mathcal{C} : \mathcal{L}(\Sigma) \rightarrow \mathcal{F}(\Sigma)$ is a **Σ -structure completing operator** $:\Leftrightarrow$
 For all structures $F \in \mathcal{L}(\Sigma) : F \subseteq \mathcal{C}(F)$ and $\mathcal{C}(F)$ is a consistent set of formulas
 and $\mathcal{C}(\rho(F)) = \rho(\mathcal{C}(F))$

with $\rho : \mathcal{F}(\Sigma) \rightarrow \mathcal{F}(\Sigma)$ being a substitution which renames object identifiers of \mathcal{V} and object set identifiers of \mathcal{W} in F without introducing name clashes; i.e. consistent renaming of identifiers and completion of formula sets are compatible operations. \square

The identity operator on sets of formulas (structures) is one trivial example of a completion operator. Any given structure is a set of atomic formulas and, therefore, free of contradictions. Furthermore, all inference systems of predicate logic (should) have the property that consistent renaming of constants does not destroy constructed proofs. Hence, we will not have any troubles with the requirement above that renaming and completion are compatible.

A useful example of a completion operation is the union of a given set of basic facts F with all derivable facts and a carefully selected subset of negated non derivable facts like “a graph does not contain a node with a name unequal to Adam” in Example 2.8. Please note that we will need an additional set of formulas for being able to derive new facts from the atomic formulas of structures. This set of first order formulas is part of the following Definition 2.9.

Example 2.8 Nonmonotonic Reasoning.

The singleton set $F := \{\text{node(Adam, man)}\}$ 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 a single node. For this purpose, the operator \mathcal{C} has to be defined as follows (omitted sets of formulas deal with edges etc.):

```

node class person;
  attribute income: integer ;
end ;
node class man is a person end ;
node class woman is a person end ;
edge type child: person → person ;
edge type wife: man [0:1] → woman [0:1] ;
path ancestor: person → person =
  ←child—
  & (self or ancestor)
end ;
path brother: person → man =
  ( ←child—
  & —child→ & instance of man )
  but not self
end ;

```

Figure 2: Excerpt from a PROGRES graph schema definition

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

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

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

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

Definition 2.9 Σ -Structure Schema.

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

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

Definition 2.10 Schema Consistent Structure.

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

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

The definition of *schema consistency* is very similar to a related definition of the knowledge representation language Telos [40]. 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 selected graph data model and excludes dangling edges and attributes of nonexisting nodes.
- The second subset contains all person database specific integrity constraints, like “man and woman nodes are also person nodes”.
- 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 $S := (\Phi, \mathcal{C})$ for graph F in Fig. 1 has the following form:

$$\begin{aligned} \Phi := & \{ \forall x, e, y : \text{edge}(x, e, y) \rightarrow \exists xl, yl : \text{node}(x, xl) \wedge \text{node}(y, yl), \\ & \forall x, a, v : \text{attr}(x, a, v) \rightarrow l : \text{node}(x, l), \dots \} \\ \cup & \{ \forall x : \text{node}(x, \text{man}) \rightarrow \text{node}(x, \text{person}), \\ & \forall x : \text{node}(x, \text{woman}) \rightarrow \text{node}(x, \text{person}), \\ & \forall x, y : \text{edge}(x, \text{wife}, y) \rightarrow \text{node}(x, \text{man}) \wedge \text{node}(y, \text{woman}), \\ & \forall x, y, z : \text{edge}(x, \text{wife}, y) \wedge \text{edge}(x, \text{wife}, z) \rightarrow y = z, \\ & \forall x, y, z : \text{edge}(x, \text{wife}, z) \wedge \text{edge}(y, \text{wife}, z) \rightarrow x = y, \\ & \forall x, v : \text{attr}(x, \text{income}, v) \rightarrow \text{type}(v, \text{integer}), \\ & \forall x : \text{node}(x, \text{person}) \rightarrow \exists v : \text{attr}(x, \text{income}, v), \dots \} \\ \cup & \{ \forall x, y : \text{ancestor}(x, y) \leftrightarrow (\exists z : \text{edge}(z, \text{child}, x) \\ & \quad \wedge (z = y \vee \text{ancestor}(z, y))), \\ & \forall x, y : \text{brother}(x, y) \leftrightarrow \exists z : \text{edge}(z, \text{child}, x) \wedge \text{edge}(z, \text{child}, y) \\ & \quad \wedge \text{node}(y, \text{man}) \wedge x \neq y, \dots \}. \end{aligned}$$

$$\mathcal{C}(F) := F \cup \dots$$

□

Figure 2 displays the corresponding PROGRES like specification of these graph properties. The first three declarations establish a node class hierarchy with *person* being the root class and *man* as well as *woman* being its subclasses. All nodes labeled with these classes have an *income* attribute. The following two declarations introduce *child* and *wife* edges and constrain the labels (classes) of their source and target nodes. Furthermore, the [0:1] annotations within the

wife edge declaration even require that any `man` node has at most one outgoing wife edge and that any `woman` node has at most one incoming edge of this type. Finally, two derived binary relations are declared, so-called *paths*, which define the well-known concepts `ancestor` and `brother`. The ancestors of a person are determined by evaluating `←child- &(self or ancestor)` as follows:

- compute the parents of a given person as the set of all sources of incoming child edges (`←child-` traverses child edges in reverse direction).
- Afterwards (`&` concatenates two subexpressions), return the union (`or`) of all parents (`self`) and their ancestors.

The brothers of a person are computed by

- determining first all its parents (`←child-`),
- finding then all male children (`&-child→ & instance of man`),
- and eliminating finally the person itself from the result (`but not self`).

In this way PROGRES allows its users to define complex graph schemata with elaborate integrity constraints as well as derived relationships (and attributes). The translation of these schema definitions into sets of formulas is always as straightforward as in the presented example above. All forthcoming rules of LBSR systems have to observe these integrity constraints, i.e. transform one schema consistent structure (graph) into another one. Furthermore, they may access derived information within pre- and postconditions.

Please note that it is not possible to construct a structure replacement machinery, where checking of integrity constraints or pre- and postconditions is guaranteed to terminate. It depends on the selected nonmonotonic reasoning strategy (the implementation of a programmed graph replacement approach) whether certain *types of constraints* are *prohibited* or lead a rule application process into an *infinite loop*. We will reconsider the termination problem later on in section 3 together with the fixpoint semantics definition for potentially nonterminating control structures.

2.2 Substructures with Additional Constraints

This subsection formalizes the term “rule match under additional constraints” by means of morphisms. This term is central to the definition of *structure replacement with pre- and postconditions*, which follows afterwards. It determines which substructures of a structure are legal targets for a structure replacement step, i.e. match a given rule’s left-hand side. The main difficulty with the definition of the new structure replacement approach was the requirement that it should include the expression oriented algorithmic graph grammar approach [43]. This approach has very powerful means for deleting, copying, and redirecting arbitrary large bundles of edges.

We will handle the *manipulation of edge bundles* by introducing special *object set identifiers* on a structure replacement rule's left- and right-hand side. These set identifiers match an arbitrarily large (maximal) set of object identifiers (see Example 2.20 and Definition 2.21). As a consequence, mappings between structures, which select the affected substructure for a rule in a structure, are neither total nor partial functions. In the general case, these *mappings are relations* between object (set) identifiers. 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 (the structure we have to manipulate) as usual and to take additional constraints for the chosen embedding into account.

Another problem comes from the fact that we have to deal with attributes, i.e. LBSR rules must be able to write, read, and modify attribute values in the same way as they are able to manipulate structural information. Therefore, it is not sufficient to map object (set) identifiers onto (sets of) object identifiers, but we need also the possibility to map them onto (sets of) attribute value representing Σ -terms.

Definition 2.12 Σ -Term Relation.

Let $F, F' \in \mathcal{F}(\Sigma)$ be two sets of formulas which have $\mathcal{V}_F, \mathcal{W}_F$ and $\mathcal{V}_{F'}, \mathcal{W}_{F'}$ as their object (set) identifiers. Furthermore, $\mathcal{T}(\Sigma)$ is the set of all variable and object identifier free Σ -terms (cf. Def. 2.3). A relation

$$u \subseteq (\mathcal{V}_F \times (\mathcal{V}_{F'} \cup \mathcal{T}(\Sigma))) \cup (\mathcal{W}_F \times (\mathcal{W}_{F'} \cup \mathcal{V}_{F'} \cup \mathcal{T}(\Sigma)))$$

is a Σ -**relation** from F to F' $:\Leftrightarrow$

- (1) For all $v \in \mathcal{V}_F : |u(v)| = 1$ with $u(x) := \{y | (x, y) \in u\}$,
i.e. every object identifier will be mapped onto a single object identifier or onto a single attribute value representing Σ -term.
- (2) For all $w \in \mathcal{W}_F : |u(w)| = 1$ or $u(w) \subseteq \mathcal{V}_{F'}$ or $u(w) \subseteq \mathcal{T}(\Sigma)$,
i.e. every object set identifier will be mapped either onto another object set identifier or onto a set of object identifiers or attribute value representing Σ -terms. \square

The definition above introduces relations between object (set) identifiers and (sets of) Σ -terms only. Their extensions to relations between arbitrary formulas are defined as follows:

Definition 2.13 Σ -Relation

Let u be a Σ -term relation according to Definition 2.12. The **extension** u^* of u to the domain of Σ -formulas is called Σ -**relation** and defined as follows.

$$\begin{aligned}
u^*(\phi, \phi') &:\Leftrightarrow \exists \psi \in \mathcal{F}(\Sigma) \text{ with free variables } x_1, \dots, x_n \\
&\quad \text{but without any } \mathcal{V} \text{ or } \mathcal{W} \text{ identifiers} \\
&\wedge \exists v_1, \dots, v_n \in \mathcal{V} \cup \mathcal{W} \text{ with } v_i \neq v_j \text{ for } i, j \in \{1, \dots, n\} \\
&\wedge \exists \tau_1, \dots, \tau_n \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{T}(\Sigma) \text{ with } u(v_1, \tau_1), \dots, u(v_n, \tau_n) : \\
&\quad \phi = \psi[v_1/x_1, \dots, v_n/x_n] \wedge \phi' = \psi[\tau_1/x_1, \dots, \tau_n/x_n].^e
\end{aligned}$$

Furthermore, with $\Phi \subseteq \mathcal{F}(\Sigma)$, the following short-hand will be used in the sequel: $u^*(\Phi) := \{\phi' \in \mathcal{F}(\Sigma) \mid \phi \in \Phi \text{ and } (\phi, \phi') \in u^*\}$. \square

The definition of u^* above simply states that two formulas ϕ and ϕ' are related to each other if they differ with respect to u -related terms only. Any object identifier v in ϕ must be replaced by the uniquely defined $u(v)$ in ϕ' . Any object set identifier w in ϕ must be replaced by some element $u(w)$ in ϕ' such that all occurrences of w in ϕ get the same substitute.

Based on these relations between formulas, we are now able to define mappings (morphisms) between sets of formulas or structures which preserve certain properties.

Definition 2.14 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: $u : F \xrightarrow{\sim} F'$) $:\Leftrightarrow F' \vdash u^*(F)$.

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

The definition of a morphism $u : F \xrightarrow{\sim} F'$ requires in the simplest case that $u^*(F)$ is a subset of F' , as in the following example:

Example 2.15 A Simple Structure Morphism

Let F' be the structure of Example 2.6 which represents the graph of Figure 1 and

$$\begin{aligned}
F &:= \{ \text{node(He, man)}, \text{node(She, woman)}, \\
&\quad \text{attr(She, income, Value)}, \text{edge(He, wife, She)}, \\
&\quad \text{edge(He, child, HisChildren)}, \text{edge(She, child, HerChildren)} \}
\end{aligned}$$

with $\text{He, She, Value} \in \mathcal{V}$, and $\text{HisChildren, HerChildren} \in \mathcal{W}$.

^e $[\rho_1/x_1, \dots, \rho_n/x_n]$ denotes consistent substitution of any x_i by its corresponding ρ_i .

There are two Σ -relations from F to F' which are structure morphisms:

$$\begin{aligned} u &:= \{ (\text{HerChildren}, \text{Hannah}), (\text{HerChildren}, \text{Charlie}), (\text{HerChildren}, \text{Fred}), \\ &\quad (\text{He}, \text{Andy}), (\text{She}, \text{Maggie}), (\text{Value}, 6000), (\text{HisChildren}, \text{Hannah}) \}, \\ u' &:= \{ (\text{HerChildren}, \text{Sally}), (\text{HerChildren}, \text{Andy}), \\ &\quad (\text{He}, \text{Adam}), (\text{She}, \text{Eve}), (\text{Value}, 10000) \}. \end{aligned} \quad \square$$

The selected examples of morphisms show that two different object (set) identifiers in F may be mapped onto the same object identifier in F' (unless explicitly prohibited by means of additional formulas). Furthermore, an object set identifier may be mapped onto an arbitrarily large set of identifiers (terms), with the empty set being a permitted special case. In the sequel, we will show that morphisms between structures define a transitive, reflexive, and associative relation, i.e. they build a category.

Proposition 2.16 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:

- (1) Σ -Morphisms are closed w.r.t. “ \circ ”, i.e.

$$u : F \xrightarrow{\sim} F', u' : F' \xrightarrow{\sim} F'' \Rightarrow (u \circ u') : F \xrightarrow{\sim} F'':$$

u and u' are Σ -relations $\Rightarrow_{\text{Def. 2.12}}$ $(u \circ u')$ is a Σ -relation and

$$\text{Def. 2.13} \quad (u \circ u')^* = u^* \circ u'^*.$$

u, u' are morphisms $\Rightarrow_{\text{Def. 2.14}}$ $F' \vdash u^*(F)$ and $F'' \vdash u'^*(F')$

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

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

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

- (2) 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,

$$id_F^*(F) = F \Rightarrow F \vdash id_F^*(F) \Rightarrow id_F : F \xrightarrow{\sim} F,$$

i.e. id_F is the required neutral morphism.

- (3) 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 $\mathcal{L}(\Sigma)$. \square

Definition 2.17 Substructure.

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

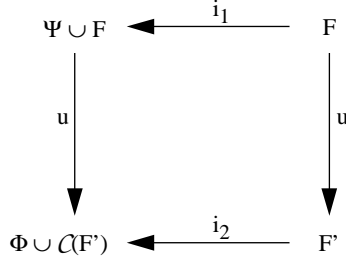


Figure 3: Substructure selection with additional constraints

This definition coincides with the usual meaning of a homomorphic^f substructure (or subgraph), if F does not contain any object set identifiers.

Proposition 2.18 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:

$$\begin{aligned}
F \subseteq_u F' &\Leftrightarrow_{\text{see Def. 2.17}} u : F \xrightarrow{\sim} F' \\
&\Leftrightarrow_{\text{see Def. 2.14}} F' \vdash u^*(F) \Leftrightarrow_{u^*(F), F' \in \mathcal{L}(\Sigma)} u^*(F) \subseteq F'.
\end{aligned}$$

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. \square

Definition 2.19 Substructure with Additional Constraints.

$S := (\Phi, \mathcal{C})$ is a Σ -structure schema, $F, F' \in \mathcal{L}(\Sigma)$, 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) $u : \Psi \cup F \xrightarrow{\sim} \Phi \cup \mathcal{C}(F')$, i.e. $\Phi \cup \mathcal{C}(F') \vdash u^*(\Psi \cup F) = u^*(\Psi) \cup u^*(F)$.

These conditions are equivalent to the existence of the diagram in Figure 3, with inclusions i_1 and i_2 being morphisms, which map any object (set) identifier onto itself:

$$\begin{aligned}
i_1 : F &\xrightarrow{\sim} \Psi \cup F, & \text{since } i_1^*(F) = F \subseteq \Psi \cup F, \text{ i.e. } \Psi \cup F \vdash i_1^*(F). \\
i_2 : F' &\xrightarrow{\sim} \Phi \cup \mathcal{C}(F'), & \text{since } i_2^*(F') = F' \subseteq \mathcal{C}(F'), \text{ i.e. } \Phi \cup \mathcal{C}(F') \vdash i_2^*(F'). \quad \square
\end{aligned}$$

^f“homomorphic” means that different object identifiers in F may be mapped onto the same object identifier in F' .

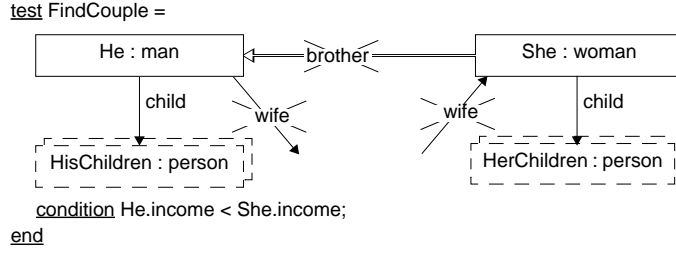


Figure 4: A subgraph test written in PROGRES

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. We may use the basic facts of F' including all formulas generated by the completing operator \mathcal{C} and the set of formulas Φ of the structure schema S for this purpose.

Example 2.20 Substructure Selection.

Let $S := (\Phi, \mathcal{C})$ be the schema of Example 2.11 and F' the database of Example 2.6. The structure $F \in \mathcal{L}(\Sigma)$ and its accompanying set of constraints $\Psi \in \mathcal{F}(\Sigma)$, defined below, represent the PROGRES like subgraph test of Figure 4 “select any pair of unmarried persons and all their children, who are not brother and sister and have appropriate income values together with all their children”:

$$\begin{aligned}
 F &:= \{ \text{node}(\text{He}, \text{man}), \text{node}(\text{She}, \text{woman}), \\
 &\quad \text{attr}(\text{He}, \text{income}, \text{HisValue}), \text{attr}(\text{She}, \text{income}, \text{HerValue}), \\
 &\quad \text{edge}(\text{He}, \text{child}, \text{HisChildren}), \text{edge}(\text{She}, \text{child}, \text{HerChildren}) \}. \\
 \Psi &:= \{ \neg \text{brother}(\text{She}, \text{He}), \neg \exists x : \text{edge}(\text{He}, \text{wife}, x), \neg \exists x : \text{edge}(x, \text{wife}, \text{She}), \\
 &\quad \text{HisValue} < \text{HerValue}, \\
 &\quad \text{node}(\text{HisChildren}, \text{person}), \text{node}(\text{HerChildren}, \text{person}) \}.
 \end{aligned}$$

Remember that He, She, and Value are object identifiers, whereas HisChildren and HerChildren are object set identifiers (cf. Example 2.2). Therefore, the following definition

$$\begin{aligned}
 u &:= \{ (\text{He}, \text{Harry}), (\text{She}, \text{Sally}), (\text{HisValue}, 10000), (\text{HerValue}, 20000), \\
 &\quad (\text{HisChildren}, \text{Linus}), (\text{HisChildren}, \text{Clio}), \\
 &\quad (\text{HerChildren}, \text{Clio}), (\text{HerChildren}, \text{Mary}) \}
 \end{aligned}$$

is a correct Σ -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 \mathcal{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 Sally (cf. definition of brother in Example 2.11)
- Harry is not the source of a wife-edge, and
- Sally is not the target of a wife-edge.

Next, `HisValue` is bound to 10000 and `HerValue` is bound to 20000, i.e. the attribute condition `HisValue < HerValue` is valid. Finally, using the formulas of the graph schema of example 2.11 we are able to prove that `Linus`, `Clio`, and `Mary` are not only direct members of the classes `man` and `woman`, respectively, but also members of the class `person`. \square

The relation u of Example 2.20 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 Σ -relations, which are involved in the process of structure rewriting.

2.3 Schema Preserving Structure Replacement

Having presented the definitions of structure schemata, schema consistent structures, and structure morphisms, we are now prepared to introduce *structure replacement rules* as quadruples of *sets of closed formulas*. The application of structure replacement rules will be defined as the construction of commuting diagrams in a similar way as it is done in the algebraic graph grammar approach [17]. But note that we use Σ -relations instead of (partial) functions. Therefore, the main properties of the algebraic approach are necessarily lost: Our (sub-)diagrams are *not pushouts* and the application of a rule is *not invertible* in the general case.

Unfortunately, we had to pay this price for the ability to formalize complex rules, which match sets of nodes and are thereby able to delete/copy/redirect arbitrarily large edge bundles.

Definition 2.21 Structure Replacement 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 replacement 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 and vice-versa; 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 rest of the modified structure. \square

The following example defines a structure replacement rule which is based on the structure selecting Example 2.20. It marries two persons under certain preconditions, whereby each person adopts the other person's children. The corresponding PROGRES like production is displayed in Figure 5.

Example 2.22 The Structure Replacement Rule Marry.

With F and Ψ defined as in Example 2.20, the structure replacement rule $\text{Marry} := (AL, L, R, AR)$ has the following form:

- (1) $AL := \Psi$.
- (2) $L := F$.
- (3) $R := F \setminus \{\text{attr}(\text{She}, \text{income}, \text{Value})\}$
 $\cup \{\text{edge}(\text{He}, \text{wife}, \text{She}),$
 $\text{attr}(\text{She}, \text{income}, \text{Value} + \text{taxReduction}(\text{Value})),$
 $\text{edge}(\text{He}, \text{child}, \text{HerChildren}), \text{edge}(\text{She}, \text{child}, \text{HisDaughters})\}$.
- (4) $AR := \{\}$. \square

Figure 6 displays the result of applying the structure replacement 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 structure replacement step are shown in Example 2.20. In the second structure replacement 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 steps are executed as follows:

- Find suitable matches for the object identifiers of L (She and He) in the database such that the corresponding preconditions in AL are satisfied.
- Extend the selected match to all object set identifiers in L and bind each set identifier to a maximal set of objects in the database such that all preconditions are valid.
- Remove the image of L without the image of R from the database (the old income attribute value of the selected She node).
- Add an image of R without the image of L to the database (the new income attribute value for She , a wife -edge between He and She , and child -edges to children nodes).

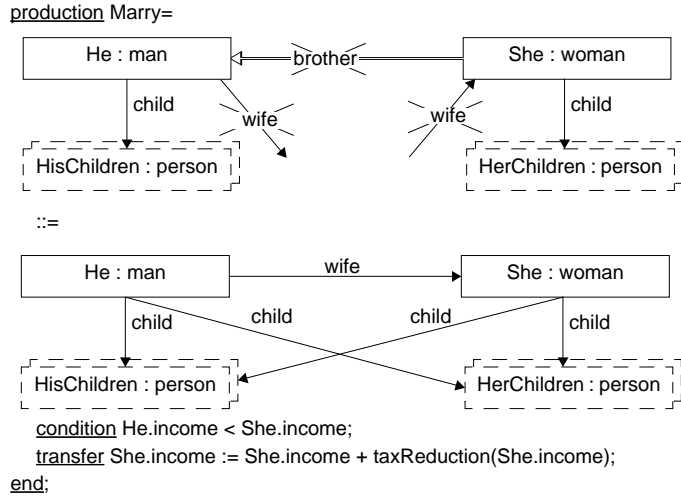


Figure 5: A graph replacement rule written in PROGRES

- Test the postconditions AR . If any postcondition is violated, then cancel all modifications, return to the first step of our list, and restart rule application with another match.
- Finally, test whether the constructed database is schema consistent and treat inconsistency like violated postconditions.

Applications of the 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, how any set of postconditions (referencing for instance derived properties of new objects) may be transformed into an equivalent set of preconditions.

Definition 2.23 Schema Preserving Structure Replacement.

$S := (\Phi, \mathcal{C}) \in \mathcal{S}(\Sigma)$ is a structure schema and $F, F' \in \mathcal{L}(S)$ are two schema consistent structures. Furthermore, $p := (AL, L, R, AR) \in \mathcal{P}(\Sigma)$ is a structure replacement rule. The structure F' is **direct derivable** from F by applying p (write: $F \xrightarrow{p} F'$) $:\Leftrightarrow$

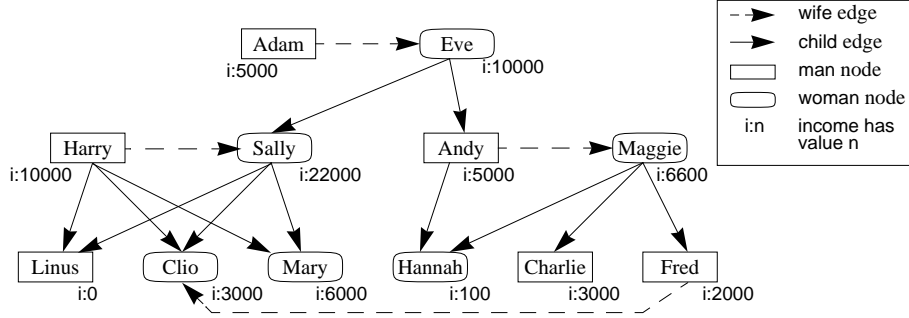


Figure 6: A modified person database

- (1) There is a morphism $u : L \xrightarrow{\sim} F$ with: $L \subseteq_{u,AL} F$,
i.e. the via u selected redex in F respects the preconditions AL .
- (2) There is no morphism $\hat{u} : L \xrightarrow{\sim} F$ 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 $w : R \xrightarrow{\sim} F'$ 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 , 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 all identifiers in K .
- (6) There exists a structure
 $H \in \mathcal{L}(\Sigma) : F \setminus (u^*(L) \setminus v^*(K)) = H = F' \setminus (w^*(R) \setminus v^*(K))$,
i.e. H represents the intermediate state after the deletion of the old substructure and before the insertion of the new substructure. \square

It is a straightforward task to transform the definition of schema preserving structure replacements 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 2.23 and is equivalent to the construction of the diagram in Figure 7. Unfortunately, we *cannot 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 replacement programs.

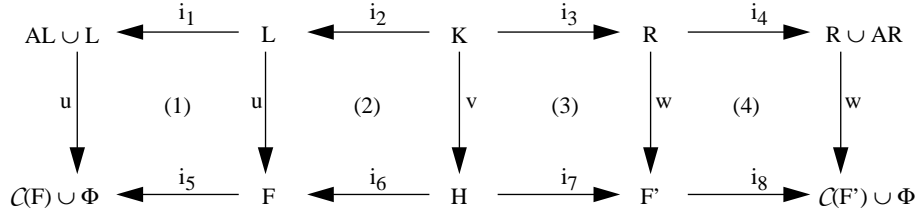


Figure 7: Diagram for the application of a structure replacement rule

Note that this kind of structure replacement does not prohibit the selection of *homomorphic* matches with two identifiers $o1$ and $o2$ in L being mapped onto the same object in F . The application of the replacement rule *Marry* above relates for instance the two set identifiers *HisChildren* and *HerChildren* to the same node *Clio* of our graph database. “Sharing” is even permitted 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 Chapter 4 of this volume). Otherwise, relation v , the restriction of u to K , would no longer be a morphism from K to H . For readability reasons, Σ -morphisms which map different node identifiers of a production onto the same node in a given graph, may, but need not be prohibited in the language *PROGRES*.

Finally note that we have to take care about *dangling references* in the general case of a person database replacement rule, where L is not a subset of R . We have to guarantee that any node (x, l) is removed together with all related $\text{edge}(x, e, y)$, $\text{edge}(y, e, x)$, and $\text{attr}(x, a, y)$ formulas. This may be accomplished by adding appropriate 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 (cf. Chapter 5 of this volume).

Proposition 2.24 Structure Replacement is Construction of Diagrams.

Assuming the terminology of Definition 2.23 and $F \xrightarrow{p} F'$ we are able to construct the diagram of fig. 7 (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 2.19. For proving the existence of commuting subdiagrams (2) and (3), we will show that step (5) of Definition 2.23 constructs a morphism v from K to H :

We start with Definition 2.23, condition (1):

$$\begin{aligned}
u : L &\xrightarrow{\sim} F \\
&\Leftrightarrow_{\text{Def. 2.14, Prop. 2.18, } K \subseteq L} u^*(K) \subseteq u^*(L) \subseteq F \\
&\Rightarrow_{\text{condition (5) of Def. 2.23}} v^*(K) \subseteq u^*(L) \subseteq F \\
&\Rightarrow_{\text{simple transformation}} v^*(K) \setminus (u^*(L) \setminus v^*(K)) \subseteq F \setminus (u^*(L) \setminus v^*(K)) \\
&\Leftrightarrow_{\text{simple transformation}} v^*(K) \subseteq F \setminus (u^*(L) \setminus v^*(K)) \\
&\Leftrightarrow_{\text{condition (6) of Def. 2.23}} v^*(K) \subseteq H \\
&\Leftrightarrow_{\text{Def. 2.14 and Prop. 2.18}} v : K \xrightarrow{\sim} H .
\end{aligned}$$

The rest of the proof follows directly from

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

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

Being equipped with a definition of structure replacement we are now able to define LBSR systems and their generated languages. As usual, generated isomorphic language elements are identified, i.e. *concrete* Σ -structures according to Definition 2.5 are replaced by equivalent classes of concrete Σ -structures, so-called *abstract* Σ -structures.

Definition 2.25 Abstract Σ -Structure.

The set of **abstract** Σ -structures $\mathcal{AL}(\Sigma) := \mathcal{L}(\Sigma) / \cong$ consists of all equivalence classes of Σ -structures with respect to the following equivalence relation \cong :

$$\forall F, F' \in \mathcal{L}(\Sigma) : F \cong F' :\Leftrightarrow \exists \text{ isomorphism } u : F \xrightarrow{\sim} F' . \quad \square$$

It is worth-while to notice that any isomorphism between two Σ -structures is a *bijective function* which maps object identifiers onto object identifiers and object set identifiers onto object set identifiers (cf. Def. 2.12, 2.13, and 2.14 of Σ -relations and Σ -morphisms). Otherwise, F would contain less object (set) identifiers than F' or the other way round. As a consequence, it would not be possible to construct the left- and right-inverse morphism u^{-1} for u . This line of argumentation is no longer true, when arbitrary sets of Σ -formulas are regarded. A single *nonatomic formula* then represents a set of derivable facts. It is, therefore, possible that two sets of formulas are equivalent (isomorphic) although they differ with respect to the number of used object (set) identifiers.

Definition 2.26 Abstract Schema Consistent Σ -Structure.

Let $S \in \mathcal{S}(\Sigma)$ be a schema and $\mathcal{L}(S)$ the set of all S -schema consistent structures. The set of **abstract schema consistent** Σ -structures is defined as

$$\mathcal{AL}(\Sigma) := \mathcal{L}(S) / \cong . \quad \square$$

Proposition 2.27 Soundness of Abstract Schema Consistent Σ -Structures.

Let $S := (\Phi, \mathcal{C}) \in \mathcal{S}(\Sigma)$ be a schema and $F, F' \in \mathcal{L}(\Sigma)$ arbitrary Σ -structures.

Then

$$F \cong F' \Rightarrow (F \in \mathcal{L}(S) \Leftrightarrow F' \in \mathcal{L}(S)),$$

i.e. all Σ -structures of an equivalence class in $\mathcal{AL}(\Sigma)$ are either schema consistent or inconsistent.

Proof:

$$F \cong F'$$

$$\Rightarrow_{\text{Def. 2.25}} \exists \text{ isomorphism } u : F \xrightarrow{\sim} F', \text{ a bijective function}$$

$$\Rightarrow_{\text{Props. 2.18}} u^*(F) = F'$$

Therefore, the following is true:

$$\begin{aligned} F \in \mathcal{L}(S) &\Rightarrow_{\text{Def. 2.10}} \mathcal{C}(F) \cup \Phi \text{ is a consistent set of formulas} \\ &\Rightarrow u^*(\mathcal{C}(F) \cup \Phi) \text{ is a consistent set of formulas} \\ &\quad (\text{consistent renaming preserves proofs}) \\ &\Rightarrow_{\text{Def. 2.9}} u^*(\mathcal{C}(F)) \cup \Phi \text{ is a consistent set of formulas} \\ &\quad (\Phi \text{ has no identifiers of } \mathcal{V} \text{ or } \mathcal{W} \Rightarrow u^*(\Phi) = \Phi) \\ &\Rightarrow_{\text{Def. 2.7}} \mathcal{C}(u^*(F)) \cup \Phi \text{ is a consistent set of formulas} \\ &\Rightarrow \mathcal{C}(F') \cup \Phi \text{ is a consistent set of formulas} \\ &\Rightarrow F' \in \mathcal{L}(S). \end{aligned}$$

In a similar way, we can prove that $F' \in \mathcal{L}(S) \Rightarrow F \in \mathcal{L}(S)$. \square

Definition 2.28 Logic-Based Structure Replacement (LBSR) System.

A tuple $\text{LBSR} := (AS, P)$ is a **logic-based structure replacement system** with respect to a structure schema $S \in \mathcal{S}(\Sigma)$ if:

- (1) $AS \in \mathcal{AL}(S)$ is the schema consistent initial abstract structure.
- (2) $P \subseteq \mathcal{P}(\Sigma)$ is a set of structure replacement rules. \square

Definition 2.29 Generated Language of LBSR System.

With $\text{LBSR} := (AS, P)$ as in Definition 2.28, the **language** $\mathcal{AL}(\text{LBSR}) \subseteq \mathcal{AL}(S)$ is defined as follows:

$$AF \in \mathcal{AL}(\text{LBSR}) :\Leftrightarrow AS \xRightarrow{P}^* AF$$

where

$$\xRightarrow{P}^* \text{ is the transitive, reflexive closure of } \xRightarrow{P}$$

and $AF \xRightarrow{P} AF' \subseteq \mathcal{AL}(\Sigma) \times \mathcal{AL}(S) :\Leftrightarrow$

$$\exists F \in AF, F' \in AF', p \in P : F \xRightarrow{p} F'. \quad \square$$

The definition above states that the *language of a LBSR system* consists of all those abstract structures which may be generated by applying its rules an arbitrary number of times to the initial abstract structure AS . A distinction between terminal and nonterminal structures has not been made in the preceding subsections. Hence, any generated abstract structure is per definition an element of the generated language.

2.4 Summary

The definitions and propositions of the preceding subsections constitute a *framework* for the formal treatment of various forms of graph replacement systems as special cases of logic-based structure replacement (LBSR) systems. Other approaches with the same intention are “**H**igh **L**evel **R**eplacement (HLR) systems” and “structured graph grammars”. HLR *systems* belong to the algebraic branch of graph grammars (cf. Section 5 of Chapter 5 and [16]). They provide a very general framework for the definition of new kinds of replacement systems, which is based on the construction of categories with certain properties. Therefore, HLR systems are not restricted to the manipulation of graphs or relational structures (as LBSR systems are).

Structured graph grammars [33], on the other hand, are restricted to a data model of directed acyclic graphs. They were a first attempt to combine properties of algebraic graph grammars (cf. Chapter 4 and 5 of this volume) and algorithmic node replacement graph grammars (cf. Chapter 2 of this volume) within a single framework. Both HLR systems and structured graph grammars do not introduce the notion of a *schema* and do not support modeling of derived properties or constraints.

LBSR systems are an attempt to close the gap between the *operation-oriented* manipulation of data structures by means of 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 profit from each other:

- Structure (graph) replacement rules might be a very comfortable and well-defined mechanism for manipulating knowledge bases or deductive data bases (cf. [23,57]), whereas
- many logic-based techniques have been proposed for efficiently maintaining derived properties of data structures, solving constraint systems, and for proving the correctness of data manipulations (cf. [8,27,52]).

Currently, LBSR systems are restricted to *sequential graph replacement*. It is subject to future work to extend the presented approach such that *parallel application of rules* is supported. Furthermore, the following questions should be considered in more detail:

- (1) Which restrictions are sufficient to guarantee that subdiagrams (2) and (3) of Figure 7 are pushouts?
- (2) Can we characterize a “useful” subset of rules and consistency constraints such that an effective proof procedure exists for the “are all derivable structures schema consistent” problem (see also Chapter 1 of this volume and [11])?

- (3) Can we develop a general procedure which transforms any set of structure replacement 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 [8]. 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 set of preconditions of a given rule and the set of guaranteed integrity constraints (for the input of the rule).

3 Programmed Structure Replacement Systems

So far we would be able to define sets of schema consistent structure replacement rules and structure languages, which are generated by applying these 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 replacement systems for specification purposes additional means are necessary for *regulating the application of rules*. Many quite different proposals for controlling application of rules may be found in literature as for instance [9,13,42,43,62]:

- Apply rules as long as appropriate or as long as possible in any order; this is the *standard semantics* of replacement systems.
- Introduce *rule priorities* and prefer applicable rules with higher priorities at least in the case of overlapping matches.
- Use *regular expressions* or even complex programs to define permissible derivation sequences.
- Draw *control flow graphs* and interpret them as graphical representations of rule controlling programs.

The idea of using *programs for controlling the application of rules*, i.e. imperative control structures, seems to be the most popular one and even superior to almost all other rule regulation approaches with respect to their expressiveness. On the following pages we will, therefore,

- first study required properties of graph replacement programs or, more general, of control structures for structure replacement rules (Subsection 3.1),
- introduce a more or less minimal set of control structures in the form of so-called **Basic Control Flow** (BCF) operators and discuss their intended semantics on an informal level (Subsection 3.2),
- present an appropriate semantic domain for BCF operators and a recently developed fixpoint theorem (Subsection 3.3),

- define the semantics of BCF expressions with recursion and, thereby, the semantics of **P**rogrammed **L**ogic-based **S**tructure **R**eplacement (PLSR) systems by means of the presented fixpoint theorem (Subsection 3.4),
- and finally summarize the main properties of PLSR systems and discuss possible extensions (Subsection 3.5).

3.1 Requirements for Rule Controlling Programs

So-called *programmed graph grammars* were already suggested many years ago in [9,43]. Nowadays, they are a fundamental concept of the graph grammar specification languages PAGG [24,26] and PROGRES [64,69]. Experiences with using these languages and their predecessors for the specification of software engineering tools [19,20,36,73] showed that their control structures should possess the following properties:

- (1) *Boolean nature*: the application of a programmed graph transformation either succeeds or fails like the application of a single graph replacement rule and, depending on success or failure, execution may proceed along different control flow paths.
- (2) *Atomic character*: a programmed sequence of graph replacement steps modifies a given host graph if and only if all its intermediate replacement steps do not fail.
- (3) *Consistency preserving*: programmed graph replacement has to preserve a graph's consistency with respect to a given set of separately defined integrity constraints.
- (4) *Nondeterministic behavior*: the nondeterminism of a single rule — it replaces any match of its left-hand side — should be preserved as far as possible on the level of programmed graph transformations.
- (5) *Recursive definition*: for reasons of convenience as well as expressiveness, programmed graph transformations should be allowed to call each other without any restrictions including any kind of recursion.

Without conditions (1) through (4) we would have difficulties to replace a complex graph replacement rule by an equivalent sequence of simpler rules, and without condition (5) the manipulation of recursively defined data structures would be quite cumbersome.

In the sequel, programmed graph transformations with the above mentioned properties will be termed *transactions*. The usage of the term “transaction” underlines the most important properties of programmed transformations: *atomicity* and *consistency*. The usually mentioned additional properties of transactions, *isolation* and *duration*, are irrelevant as long as parallel programming is not supported and backtracking may cancel the effects of already successfully completed transactions (cf. Subsection 3.2).

```

<Transaction> ::=      <TransactionId> "=" <BCFExpr> ;
<BCFExpr> ::=         <BasicAction> | <ActionCall> | <BCFTerm> ;
<BasicAction> ::=    "skip" | "loop" ;
<ActionCall> ::=     <RuleId> | <TransactionId> ;
<BCFTerm> ::=        "def" "(" <BCFExpr> ")" | "undef" "(" <BCFExpr> ")" |
                    "(" <BCFExpr> ";" <BCFExpr> ")" |
                    "(" <BCFExpr> "[]" <BCFExpr> ")" |
                    "(" <BCFExpr> "&" <BCFExpr> ")";

```

Figure 8: Syntax of graph transactions and BCF expressions

3.2 Basic Control Flow Operators

The definition of a fixed set of control structures for structure manipulating transactions is complicated by contradicting requirements. From a *theoretical point of view* the set of offered control structures should be as small as possible, and we should be allowed to combine them without any restrictions. But from a *practical point of view* control structures are needed which are easy to use, which cover all frequently occurring control flow patterns, and the application of which may be directed by a number of context-sensitive rules.

Therefore, it was quite natural to distinguish between basic control flow operators, in the sequel termed *BCF operators*, of an underlying theory of recursively defined transactions and more or less complex higher level programming mechanisms. Starting with a formal definition of basic control flow operators, we should then be able to define the meaning of a large class of programming mechanisms by translating them into equivalent BCF expressions (cf. Section 5).

Figure 8 contains the definition of BCF expressions themselves and of transactions as functional abstractions of BCF expressions. It distinguishes between *basic actions* like

- skip, which represents the always successful identity operator and relates a given graph G to itself, and
- loop, which neither succeeds nor terminates for any given structure and represents therefore “crashing” or forever looping computations,

calls of simple rules or other graph transactions, and finally between two unary and three binary *BCF operators* with

- def(a) as an action which succeeds applied to a given structure F , whenever a applied to F produces a defined result, and returns F itself,
- undef(a) as an action which succeeds applied to a structure F , whenever a applied to F terminates with failure, and returns F itself,

- $(a; b)$ as 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 a ,
- $(a \parallel b)$ as an action which represents the nondeterministic choice between the application of a or b , and
- $(a \& b)$ as an action which returns the intersection of the results of a and b (requires an equality or isomorphy testing operator on graphs).

Note that the operators suggested above are intentionally similar to those proposed by Dijkstra [12] and especially to those presented by Nelson [49] with one essential difference: due to the boolean nature of basic structure replacement rules and complex transactions, we are not forced to distinguish between side effect free boolean expressions and state modifying actions. This has the consequence that complex guarded commands of the form

$$(\text{Cond}_1 \rightarrow \text{Body}_1 \parallel \dots \parallel \text{Cond}_n \rightarrow \text{Body}_n)$$

are no longer necessary but may replaced by expressions like

$$(\underline{\text{def}}(\text{Cond}_1); \text{Body}_1) \parallel \dots \parallel (\underline{\text{def}}(\text{Cond}_n); \text{Body}_n)$$

where $\underline{\text{def}}(\text{Cond}_i)$ tests either the applicability of a single rule or of a whole transaction without modifying the given input structure.

Furthermore, BCF expressions offer almost all possibilities for combining binary relations over structures, the semantic domain of structure manipulating transactions. There are operators for intersection, union, and concatenation. The *missing difference operator* is not supported for the following reasons: $a \setminus b1$ is a subrelation of $a \setminus b2$, if $b2$ is a subrelation of $b1$. As a consequence, the nonmonotonic difference operator had to be excluded in order to be able to come up with a sound definition for recursively defined transactions (cf. Subsection 3.4).

Having motivated our reasons for the selection of two unary and three binary 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 transformation a to a structure F has three possible results named $F1$, $F2$, and $F3$, respectively. Furthermore, let us assume that the graph transformation b applied to $F1$ fails but applied to $F2$ and $F3$ succeeds. In this case we may either select $F2$ or $F3$, but not $F1$ 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 a more realization-oriented definition of this kind of clairvoyant nondeterminism requires a kind of *depth-first search* semantics 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 structure replacement rules but not selections which cause nonterminating computations.

The following example uses nondeterministic depth-first search and backtracking. It defines a transaction, which tries to complete a person database such that all person nodes are married afterwards. It uses the rule `Marry` of Example 2.22 and assumes the existence of another rule `MarkUnmarried`. The rule `MarkUnmarried` selects and marks a single unmarried person in the database if existent and fails otherwise.

Example 3.1 A Person Database Manipulating Transaction.

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

The transaction above initiates calls of rule `Marry` as long as the rule `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).

3.3 Preliminary Definitions

After an informal introduction of transactions as named BCF expressions we will now define their intended semantics: This is a *semantic function* from the domain of BCF expressions onto the range of (extended) binary relations over abstract structures. In order to be able to deal with recursion and nondeterminism in the presence of an atomic sequence operator “;” we had to follow

the lines of [49]⁹. There, a new form of the fixpoint theorem is used to give an axiomatic definition of so-called nondeterministic commands.

Proposition 3.2 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 [49]. □

For being able to apply the fixpoint theorem to BCF expressions an appropriate partially ordered semantic domain is needed:

Definition 3.3 Semantic Domain.

With $\mathcal{AL}(S)$ being a S -consistent class of abstract 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:

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

The semantics of a transaction is a binary relation between abstract 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 may abort or loop forever. A relation $R_\infty := \mathcal{AL}(S) \times \{\infty\}$ 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 3.4 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:

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

⁹The sequence operator is not chain continuous in its 2nd argument. Therefore, the original version of the fixpoint theorem, proved in [39], does not work in our case.

Please note that

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

is an equivalent definition of the partial order “ \leq ”. It is easier to handle and will be used from now on. \square

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 proof its *chains have joins* condition:

Lemma 3.5 Chains Have Joins.

Let $(R^\alpha)_{\alpha \in \text{ordinal}} \subseteq \mathcal{D}$ be a chain, i.e. for any ordinals a and b 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 that R is the least element in \mathcal{D} with this property:

$\forall \alpha : R^\alpha \leq R :$

$$(F, F') \in R^\alpha \Rightarrow (F, F') \in R \vee F' = \infty . \\ (F, \infty) \notin R^\alpha \Rightarrow \forall \beta \geq \alpha : (F, F') \in R^\alpha \Leftrightarrow (F, F') \in R^\beta \\ \Rightarrow (F, F') \in R^\alpha \Leftrightarrow (F, F') \in R .$$

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

$$(F, F') \in R \Rightarrow \exists R^\alpha : (F, F') \in R^\alpha \vee F' = \infty \\ \Rightarrow (F, F') \in R' \vee F' = \infty . \\ (F, \infty) \notin R \Rightarrow \exists R^\alpha : (F, \infty) \notin R^\alpha \\ \Rightarrow \exists R^\alpha : (F, F') \in R \Leftrightarrow (F, F') \in R^\alpha \Leftrightarrow (F, F') \in R' . \quad \square$$

3.4 A Fixpoint Semantics for Transactions

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 3.6 BCF Expressions, Transactions, and their Semantics.

\mathcal{S} and \mathcal{D} are defined as in Definition 3.4, and $P \subseteq \mathcal{P}(\Sigma)$ is a set of structure replacement rules. Then, $\mathcal{E}(P)$ denotes the set of all **BCF expressions** and $\mathcal{T}(P)$ the set of all **transactions** over P . Their context-free syntax is displayed

in Figure 8, and a **semantic function** $\mathcal{R}_P : \mathcal{E}(P) \rightarrow \mathcal{D}$ is defined as follows^h with abstract structures $F, F', F'' \in \mathcal{AL}(S)$ and with $a, b \in \mathcal{E}(P)$:

- (1) $(F, F') \in \mathcal{R}_P[\text{skip}] : \Leftrightarrow F = F'$.
- (2) $(F, F') \in \mathcal{R}_P[\text{loop}] : \Leftrightarrow F' = \infty$.
- (3) $(F, F') \in \mathcal{R}_P[p] : \Leftrightarrow F \xrightarrow{p} F'$, for any production $p \in \mathcal{P}(\Sigma)$
 $\vee F' = \infty$, if execution of p may not terminate.
- (4) $(F, F') \in \mathcal{R}_P[\text{def}(a)] : \Leftrightarrow \exists F'' \neq \infty : (F, F'') \in \mathcal{R}_P[a] \wedge F = F'$
 $\vee (F, \infty) \in \mathcal{R}_P[a] \wedge F' = \infty$.
- (5) $(F, F') \in \mathcal{R}_P[\text{undef}(a)] : (\neg \exists F'' : (F, F'') \in \mathcal{R}_P[a]) \wedge F = F'$
 $\vee (F, \infty) \in \mathcal{R}_P[a] \wedge F' = \infty$.
- (6) $(F, F') \in \mathcal{R}_P[(a; b)] : \Leftrightarrow \exists F'' \neq \infty : (F, F'') \in \mathcal{R}_P[a] \wedge (F'', F') \in \mathcal{R}_P[b]$
 $\vee (F, \infty) \in \mathcal{R}_P[a] \wedge F' = \infty$.
- (7) $(F, F') \in \mathcal{R}_P[(a \parallel b)] : \Leftrightarrow (F, F') \in \mathcal{R}_P[a] \vee (F, F') \in \mathcal{R}_P[b]$.
- (8) $(F, F') \in \mathcal{R}_P[(a \& b)] : \Leftrightarrow F' = \infty \wedge ((F, \infty) \in \mathcal{R}_P[a] \vee (F, \infty) \in \mathcal{R}_P[b])$
 $\vee (F, F') \in \mathcal{R}_P[a] \wedge (F, F') \in \mathcal{R}_P[b]$. \square

The definitions above are rather straightforward with the exception of the treatment of the operators **def** and **undef**. The expressions **def**(a) and **undef**(a) loop forever if a returns not a single defined result but loops forever. Furthermore, **def**(a) may return its input if a returns at least one defined result, even if a may loop forever. It terminates with failure if a terminates with failure. On the other hand, **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 .

Nevertheless, a *strict version* of **def** might be useful, too. It is no longer independent from the definition of **undef**, but may be given as follows:

^hThe definition of the semantics of transactions themselves and calls to transactions within BCF expressions has to be postponed, until the precondition of the previously introduced fixpoint theorem are checked.

Definition 3.7 Strict Version of def Operator.

A strict version of the `def` operator may be defined as follows with abstract structures $F, F', F'' \in \mathcal{AL}(S)$, and $a \in \mathcal{E}(P)$ as in Definition 3.6:

$$\begin{aligned}
(F, F') \in \mathcal{R}_P[\underline{\text{def}}^\infty(a)] &: \Leftrightarrow \\
&(F, F') \in \mathcal{R}_P[\underline{\text{undef}}(\underline{\text{undef}}(a))] \\
\Leftrightarrow &(\neg \exists F'' : (F, F'') \in \mathcal{R}_P[\underline{\text{undef}}(a)]) \wedge F = F' \\
&\vee (F, \infty) \in \mathcal{R}_P[\underline{\text{undef}}(a)] \wedge F' = \infty \\
\Leftrightarrow &(\exists F'' \neq \infty : (F, F'') \in \mathcal{R}_P[a] \wedge (F, \infty) \notin \mathcal{R}_P[a] \wedge F = F') \\
&\vee (F, \infty) \in \mathcal{R}_P[a] \wedge F' = \infty. \quad \square
\end{aligned}$$

In order to be able to use Proposition 3.2 for the treatment of recursive transactions 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:

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

Provided with the semantics $\mathcal{R}_P[t_1], \dots, \mathcal{R}_P[t_n]$ of t_1 to t_n , the semantics of E is given by Definition 3.6 and is denoted as follows:

$$\mathcal{R}_P[E][\mathcal{R}_P[t_1], \dots, \mathcal{R}_P[t_n]].$$

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

$\text{Marry}^* = (((\text{def}(\text{MarkUnmarried}); \text{Marry}); \text{Marry}^*) \parallel \underline{\text{undef}}(\text{MarkUnmarried})),$
we are looking for a least element $R \in \mathcal{D}$ such that the following *fixpoint equation*

$$R = \mathcal{R}_P[\underline{\text{undef}}(\underline{\text{def}}(\text{MarkUnmarried}); \text{Marry}); \text{Marry}^*) \parallel \underline{\text{undef}}(\text{MarkUnmarried})][R]$$

holds. Proposition 3.2 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 3.8 BCF Operators are Monotonic Functionals.

With $\mathcal{D}, \mathcal{AL}(S)$, and \mathcal{R}_P from definition 3.6, $P \subseteq P(\Sigma)$, and $a, a', b \in \mathcal{E}(P)$, $\mathcal{R}_P[a] \leq \mathcal{R}_P[a']$ implies:

- (1) $\mathcal{R}_P[(a; b)] \leq \mathcal{R}_P[(a'; b)].$
- (2) $\mathcal{R}_P[(b; a)] \leq \mathcal{R}_P[(b; a')].$
- (3) $\mathcal{R}_P[(a \parallel b)] = \mathcal{R}_P[(b \parallel a)] \leq \mathcal{R}_P[(b \parallel a')] = \mathcal{R}_P[(a' \parallel b)].$
- (4) $\mathcal{R}_P[(a \& b)] = \mathcal{R}_P[(b \& a)] \leq \mathcal{R}_P[(b \& a')] = \mathcal{R}_P[(a' \& b)].$
- (5) $\mathcal{R}_P[\underline{\text{def}}(a)] \leq \mathcal{R}_P[\underline{\text{def}}(a')].$
- (6) $\mathcal{R}_P[\underline{\text{undef}}(a)] \leq \mathcal{R}_P[\underline{\text{undef}}(a')].$

Proof:

ad (1) $\mathcal{R}_P[(a; b)] \leq \mathcal{R}_P[(a'; b)]$:

$\forall F \in \mathcal{AL}(S)$ with $(F, \infty) \notin \mathcal{R}_P[a]$

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

$\Rightarrow (F, F') \in \mathcal{R}_P[(a; b)] \Leftrightarrow (F, F') \in \mathcal{R}_P[(a'; b)]$.

$\forall F \in \mathcal{AL}(S)$ with $(F, \infty) \in \mathcal{R}_P[a] \Rightarrow (F, \infty) \in \mathcal{R}_P[(a; b)]$

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

$(F, F') \in \mathcal{R}_P[(a; b)]$

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

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

$\Rightarrow (F, F') \in \mathcal{R}_P[(a'; b)] \vee F' = \infty$.

ad (2) $\mathcal{R}_P[(b; a)] \leq \mathcal{R}_P[(b; a')]$:

$\forall F \in \mathcal{AL}(S)$ with $\neg \exists F'' : ((F, F'') \in \mathcal{R}_P[b] \wedge (F'', \infty) \in \mathcal{R}_P[a])$

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

$(F, F'') \in \mathcal{R}_P[b]$

$\Rightarrow (F'', \infty) \notin \mathcal{R}_P[a]$

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

$\Rightarrow (F, F') \in \mathcal{R}_P[(b; a)] \Leftrightarrow (F, F') \in \mathcal{R}_P[(b; a')]$.

$\forall F \in \mathcal{AL}(S)$ with $\exists F'' : ((F, F'') \in \mathcal{R}_P[b] \wedge (F'', \infty) \in \mathcal{R}_P[a])$

$\Rightarrow (F, \infty) \in \mathcal{R}_P[(b; a)]$

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

$(F, F') \in \mathcal{R}_P[(b; a)]$

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

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

$\Rightarrow \mathcal{R}_P[(b; a')] \vee F' = \infty$.

ad (3) $\mathcal{R}_P[(a \parallel b)] \leq \mathcal{R}_P[(a' \parallel b)]$:

$\forall F \in \mathcal{AL}(S)$ with $(F, \infty) \notin \mathcal{R}_P[a]$

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

$\Rightarrow (F, F') \in \mathcal{R}_P[(a \parallel b)] \Leftrightarrow (F, F') \in \mathcal{R}_P[(a' \parallel b)]$.

$\forall F \in \mathcal{AL}(S)$ with $(F, \infty) \in \mathcal{R}_P[a] \Rightarrow (F, \infty) \in \mathcal{R}_P[(a \parallel b)]$

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

$(F, F') \in \mathcal{R}_P[(a \parallel b)]$

$\Rightarrow (F, F') \in \mathcal{R}_P[a] \vee (F, F') \in \mathcal{R}_P[b]$

$\Rightarrow (F, F') \in \mathcal{R}_P[a'] \vee (F, F') \in \mathcal{R}_P[b] \vee F' = \infty$

$\Rightarrow (F, F') \in \mathcal{R}_P[(a' \parallel b)] \vee F' = \infty$.

ad (4) $\mathcal{R}_P[(a \& b)] \leq \mathcal{R}_P[(a' \& b)]$:

$\forall F \in \mathcal{AL}(S)$ with $(F, \infty) \notin \mathcal{R}_P[a]$

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

$\Rightarrow (F, F') \in \mathcal{R}_P[(a \& b)] \Leftrightarrow (F, F') \in \mathcal{R}_P[(a' \& b)]$.

$\forall F \in \mathcal{AL}(S)$ with $(F, \infty) \in \mathcal{R}_P[a] \Rightarrow (F, \infty) \in \mathcal{R}_P[(a \& b)]$

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

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

ad (5) $\mathcal{R}_P[\underline{\text{def}}(a)] \leq \mathcal{R}_P[\underline{\text{def}}(a')]$:

$$\begin{aligned} \forall F \in \mathcal{AL}(S) \text{ with } (F, \infty) \notin \mathcal{R}_P[a] \\ \Rightarrow & (F, F') \in \mathcal{R}_P[a] \Leftrightarrow (F, F') \in \mathcal{R}_P[a'] \\ \Rightarrow & (F, F') \in \mathcal{R}_P[\underline{\text{def}}(a)] \Leftrightarrow (F, F') \in \mathcal{R}_P[\underline{\text{def}}(a')]. \\ \forall F \in \mathcal{AL}(S) \text{ with } (F, \infty) \in \mathcal{R}_P[a] \Rightarrow (F, \infty) \in \mathcal{R}_P[\underline{\text{def}}(a)] \\ \text{and with } (F, F'') \in \mathcal{R}_P[a] \Rightarrow (F, F'') \in \mathcal{R}_P[a'] \vee F'' = \infty : \\ & (F, F') \in \mathcal{R}_P[\underline{\text{def}}(a)] \\ \Rightarrow & (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}_P[a] \wedge F = F'') \vee F' = \infty \\ \Rightarrow & (\exists F'' \neq \infty : (F, F'') \in \mathcal{R}_P[a'] \wedge F = F'') \vee F' = \infty \\ \Rightarrow & (F, F') \in \mathcal{R}_P[\underline{\text{def}}(a')] \vee F' = \infty. \end{aligned}$$

ad (6) $\mathcal{R}_P[\underline{\text{undef}}(a)] \leq \mathcal{R}_P[\underline{\text{undef}}(a')]$:

$$\begin{aligned} \forall F \in \mathcal{AL}(S) \text{ with } (F, \infty) \notin \mathcal{R}_P[a] \\ \Rightarrow & (F, F') \in \mathcal{R}_P[a] \Leftrightarrow (F, F') \in \mathcal{R}_P[a'] \\ \Rightarrow & (F, F') \in \mathcal{R}_P[\underline{\text{undef}}(a)] \Leftrightarrow (F, F') \in \mathcal{R}_P[\underline{\text{undef}}(a')]. \\ \forall F \in \mathcal{AL}(S) \text{ with } (F, \infty) \in \mathcal{R}_P[a] \text{ implies:} \\ & (F, F') \in \mathcal{R}_P[\underline{\text{undef}}(a)] \Leftrightarrow F' = \infty \\ \Rightarrow & (F, F') \in \mathcal{R}_P[\underline{\text{undef}}(a)] \Leftrightarrow ((F, F') \in \mathcal{R}_P[\underline{\text{undef}}(a')] \vee F' = \infty). \quad \square \end{aligned}$$

Based on the proof above we are now able to define a fixpoint semantics for a given set of transactions which may call each other in an arbitrary manner. Please note that from a superficial point of view the treatment of n transactions instead of one requires the introduction of a more *complex semantic domain* \mathcal{D}^n and a corresponding semantic function. This function takes n binary relations as input, which are already computed approximations of n transactions; it produces better approximations in the form of n binary relations as output. Applying fixpoint theory to this extended setting the semantics of the i -th transaction is defined as the i -th component of their common least fixpoint in \mathcal{D}^n . The extension of the semantic domain \mathcal{D} to \mathcal{D}^n — including a new partial chain continuous order — is rather straightforward. Therefore, it is omitted over here, although it is implicitly used in the following corollary:

Corollary 3.9 Fixpoint Semantics for Transactions.

Let $T := \{t_1, \dots, t_n\} \in \mathcal{T}(P)$ be a set of 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 structure replacement rules of $P \subseteq \mathcal{P}(\Sigma)$, i.e.

$$t_1 = E_1[t_1, \dots, t_n], \dots, t_n = E_n[t_1, \dots, t_n].$$

With $\mathcal{AL}(\Sigma)$ being the corresponding class of abstract schema consistent structures, the following propositions hold and define a new **semantic function** $\mathcal{R}_P^* : T \rightarrow \mathcal{D}$:

- (1) t_1, \dots, t_n have unique least fixpoints $\mathcal{R}_P^*[t_1], \dots, \mathcal{R}_P^*[t_n] \in \mathcal{D}$.
- (2) Approximations of these fixpoints may be constructed as follows:

$$R_1^0 := \mathcal{AL}(S) \times \{\infty\}; \dots; R_n^0 := \mathcal{AL}(S) \times \{\infty\},$$

$$R_1^{k+1} := \mathcal{R}_P^*[E_1][R_1^k, \dots, R_n^k]; \dots; R_n^{k+1} := \mathcal{R}_P^*[E_n][R_1^k, \dots, R_n^k],$$
 where $\mathcal{R}_P^*[E_i] : \mathcal{D}^n \rightarrow \mathcal{D}$ takes approximations for transactions t_1, \dots, t_n as input and yields a new approximation for t_i by applying Definition 3.6 to expression E_i .

Proof:

ad (1) Follows directly from Proposition 3.2 and Lemmata 3.5 and 3.8.

ad (2) The relations R_i^k are indeed approximations for t_1, \dots, t_n such that for any $i, k : R_i^0 \leq \dots \leq R_i^k \leq \mathcal{R}_P^*[t_i]$. This follows directly from Proposition 3.2 and Lemmata 3.5 and 3.8 with

$$\forall i, l : R_i^l = \bigcup_{k \leq l} R_i^k \leq \mathcal{R}_P^*[t_i] \text{ and } \bigcup_{k \in \mathbb{N}} R_i^k = \mathcal{AL}(S) \times \{\infty\}. \quad \square$$

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 such a fixpoint computing function, we are now able to define programmed structure replacement systems and their generated (abstract) structure languages.

Definition 3.10 Programmed Structure Replacement Systems.

Assuming the vocabulary of Definition 3.6, $\text{PLSR} := (S, A, P, T, t)$ is a **programmed structure replacement (PLSR) system** with respect to a signature Σ if:

- (1) $S \in \mathcal{S}(\Sigma)$ is a structure schema.
- (2) $A \in \mathcal{AL}(S)$ is the schema consistent initial structure.
- (3) $P \subseteq \mathcal{P}(\Sigma)$ is a set of structure replacement rules.
- (4) $T \subseteq \mathcal{T}(P)$ is a set of (recursively) defined transactions over P .
- (5) $t \in T$ is the *main transaction* of PLSR. □

Definition 3.11 Language of a Programmed Structure Replacement System.

With $\text{PLSR} := (S, A, P, T, t)$ as in Def. 3.11, the language $\mathcal{AL}(\text{PLSR})$ is defined as follows:

$$F \in \mathcal{AL}(\text{PLSR}) :\Leftrightarrow (A, F) \in \mathcal{R}_P^*[t] \wedge F \neq \infty. \quad \square$$

The definition above states that the language of a programmed structure replacement 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 replacement rules from P and other transactions from T , which in turn contain calls of rules and transactions.

In the case where PLSR systems are used to define abstract data types instead of structure (graph) languages only, it might be useful to replace the single main transaction “ t ” above by a set of *exported transactions* t_1, \dots, t_n . This would be a very first step towards a module concept for structure replacement systems. But even in that case the language of all possibly generated structures is definable as the result of the following new main transaction:

$$t' = ((t_1 \parallel \dots \parallel t_n); t') \parallel \underline{\text{skip}},$$

which either returns its input or applies the transactions t_1, \dots, t_n an arbitrary number of times.

3.5 Summary

The definitions and propositions of the preceding subsections constitute a *framework* for defining and comparing various approaches which support programming with (graph) replacement rules. The framework provides us with a formal definition of partial, nondeterministic, and even recursive *structure replacement 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. [64] 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 [49], 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 replacement rules play here about the same role as simple assignments in [49]. The definition of their intended semantics as binary relations is given in Definition 2.23, but it is a yet unsolved problem (in the general case) how to push postconditions through rules for obtaining their corresponding weakest preconditions (cf. list of unsolved question in Subsection 2.4).

Furthermore note that BCF expressions as well as almost all related approaches discussed in Section 5 deal with *sequential replacement processes* only.

The so-called *programmed derivation of relational structures* approach [42] is the only exception. This approach supports *parallel programming with graph replacement rules* and provides a fixpoint semantics for recursive programs based on program traces. Unfortunately, no constructs (like `def` and `undef` over here) are available for testing whether a complex structure transformation returns a defined result or not. Parallel composition of subtransformations as well as testing their applicability 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 non-terminating 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.

Finally, we have to emphasize that all presented results in this section are valid for *any rule-based approach*, where rules define binary relations over a given domain of objects. Having developed such a common framework for rather different rule regulation mechanisms offers the opportunity to *combine different regulation mechanisms* as needed within future graph grammar-based specification languages. This idea is closely related to the *GRACE initiative* [34], a first attempt to combine different graph models, graph replacement approaches, and rule regulation mechanisms within a single **GRA**ph **CE**ntered language and environment. As far as we can see, PLSR systems could act as the common underlying formalism for all envisaged types of graphs, rules, and control structures.

In GRACE, a complete specification is built by defining and composing so-called *transformation units*. Any transformation unit may use a different style of graph replacement (cf. section 4 and 5 for an overview about currently existing variants of graph replacement approaches). It defines a binary relation between certain classes of graphs. The operators which combine single transformation units to more complex transformation units are very similar to BCF operators presented over here. Hence, PLSR systems can be used to define the semantics of transformation units and their composition operators. The other way round, transformation units of GRACE provide a kind of *module concept* for structuring PLSR systems into reusable subcomponents with well-defined interfaces between them.

4 Context-sensitive Graph Replacement Systems — An Overview

Until now, we presented a very general framework for the definition of specific graph data models and especially for the definition of various types of (programmed) graph replacement systems. We will use this framework over here for reviewing a number of different context sensitive *graph replacement approaches* (cf. Table 1 in Subsection 4.5). These are the expression-oriented algorithmic graph replacement approach (EXP) of [43,44], and its afore-mentioned successors PAGG [25,26] and PROGRES [63,64]. These approaches will be compared with the two families of algebraic graph replacement systems, DPO [17,18] and SPO [18,38], which are the main subject of Chapter 4 and 5 of this volume. Finally, we have selected so-called DELTA grammars [31], which are very similar to PAGG, but claim to be a member of the algebraic branch.

It is rather obvious, how different graph data models — like directed graphs or hypergraphs — may be seen as relational structures with certain properties. Furthermore, all presented graph replacement systems have about the *same data model* of a directed graph with labeled nodes and edges. The only differences are that

- one approach (EXP) does not support (node) attributes,
- algebraic DPO and SPO approaches handle edges as identifiable objects and are easily extendible to n-ary relations in the form of hyperedges,
- and every second approach knows the concept of a label hierarchy, where more general node or edge labels match more specific node or edge labels.

Therefore, we will omit a more detailed discussion of graph data model differences in the sequel and simply assume the same class of graphs as in the running example of Section 2. Furthermore, we will also omit the subject of graph schema definitions, which is already covered by Subsection 2.1. But even then, a bewildering *long list of characteristics* remains which distinguish the selected six graph replacement approaches:

- The following Subsection 4.1 discusses how *matches* of rules in a host graph may be restricted and how deletion of nodes and edges is handled.
- Subsection 4.2 presents then two completely different ways how to specify the *embedding* of created nodes into the remaining part of the host graph.
- Subsection 4.3 shows afterwards why additional *application conditions* for rules are needed and how they are usually defined.
- Subsection 4.4 sketches then how *attributes* are handled within graph replacement systems.
- And subsection 4.5 summarizes the discussion in a table with one column for each presented approach and one row for each of the identified *25* graph replacement system *characteristics*.

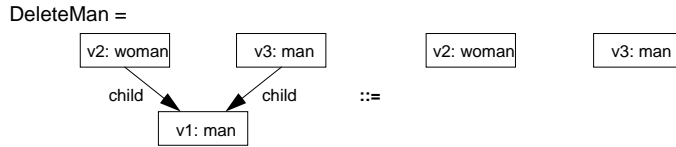


Figure 9: Example of a context-sensitive graph replacement rule

4.1 Context-sensitive Graph Replacement Rules

In the simplest case, a graph replacement rule consists of two graphs with labeled, directed edges and labeled, identifier carrying nodes. These two graphs are their left- and right-hand side. The overall idea of the *application of a graph replacement rule* is to

- (1) find first a match for its left-hand side in a given host graph,
- (2) remove then the match from the host graph,
- (3) construct an isomorphic copy of the rule's right-hand side,
- (4) and add the copy to the remaining rest of the host graph.

This procedure leads to a host graph, where the *new subgraph is isolated* from the old rest of the host graph. Therefore, an additional mechanism is needed, called *embedding transformation*, for establishing connections between new and old nodes. The most straight-forward realization of such an embedding transformation is based on the *identification of nodes* in a rule's left- and right-hand side, the so called *gluing approach*. Matches of these nodes are neither deleted nor recreated during the application of replacement rules. They simply have to be present and are preserved with all their edges to unmatched nodes in the host graph. The graph replacement rule DeleteMan of Figure 9 has for instance two nodes v2 and v3, which are part of its left- and right-hand side and whose matches in a host graph may have connections to an arbitrary number of unmatched nodes in the host graph.

Using this kind of graph replacement the following two questions arise:

- Shall we allow that the match of a rule is a nonisomorphic image of its left-hand side, i.e. that two nodes of the left-hand side share the same node in the host graph?
- And what about edges at a matched node which are not matched by an edge of the left-hand side although their node will be deleted by the rule?

All six reviewed graph replacement formalisms have different answers for these questions. The **EXP** approach [43,44] (with so-called path **EXP**ressions) does not know the concept of preserved nodes, but solves the embedding problem in a different manner (cf. Subsection 4.2). Therefore, all matched nodes are

deleted and permitting nonisomorphic images makes no sense. Furthermore, unmatched edges between matched nodes are not allowed, i.e. the matched subgraph has to be equivalent to its *induced subgraph*. As a consequence, the rule `DeleteMan` is not applicable to any man node whose parents are connected to each other with a wife edge. Edges between matched nodes and unmatched nodes are handled in a different way. They will be deleted together with their matched nodes. Otherwise, graph replacement rules would never be able to delete nodes which have connections to preserved and, therefore, unmatched nodes.

The **PAGG** (**P**rogrammed **A**ttributed **G**raph **G**rammars) approach [24,25,26] knows the concept of preserved nodes, but requires still that matches are isomorphic images of left-hand sides (with the exception of the embedding part of replacement rules which will be explained in Subsection 4.2). One of its differences to EXP is that the induced subgraph condition is dropped, i.e. all unmatched edges at a deleted node are handled in the same way and are deleted, too. Therefore, a graph replacement rule like `DeleteMan` can be used to delete any man node independent from the fact whether this node or its parent nodes are sources/targets of additional edges. Figure 10 of Subsection 4.2 contains an example for the notation of PAGG rules.

The default behavior of **PROGRES** graph replacement rules is the same as for PAGG. But explicit *folding statements* allow left-hand sides to share their matches as long as these nodes are also part of the right-hand side. This is the so-called *identification condition*, which prevents situations where a rule has to preserve and to delete a node at the same time. In the rule `DeleteMan` for instance nodes `v1` and `v3` may not match the same node in the host graph, even if the host graph contains a man node with a child edge loop.

The next approach, **DELTA** [31] (its rules have the form of a Δ), is very similar to PROGRES with respect to the supported form of graph replacement (cf. Figure 10 of Subsection 4.2). Isomorphic matching is the default, but explicit folding statements may be used to overwrite the default. Deletion/preservation conflicts are possible, but (probably) prohibited by the identification condition, too.

The **DPO** (**D**ouble **P**ush**O**ut) approach [17,18] is the most restrictive one concerning the treatment of unmatched edges at nodes which should be deleted. The so-called *dangling edge condition* forbids the application of rules in this case. Therefore, `DeleteMan` may only be used to delete man nodes without outgoing wife or child edges. Furthermore, nonisomorphic matching together with the identification condition is the default. Both the dangling edge and the identification condition are summarized under the name *gluing condition*.

The **SPO** (Single PushOut) approach [18,38], finally, permits any kind of nonisomorphic images of left-hand sides and drops also the dangling edge condition. As a consequence, edges at deleted nodes are deleted, too, and deletion/preservation conflicts are resolved in favor of deletion.

Last but not least we should mention that all discussed six approaches allow to a certain extent that a single node (edge) of a graph replacement rule matches more than one node (edge) in the host graph. At least EXP, DPO, and SPO have extensions which allow for the *parallel application* of a single rule (or of several rules) at various matches in a given host graph. Furthermore, both DPO and SPO variants were developed recently with so-called *amalgamated graph replacement rules* [7,29,70]. Such an amalgamated rule represents an infinite family of simple rules in the general case. They are useful for manipulating different occurrences of a single pattern in the host graph simultaneously. Simplified forms of amalgamation — mainly used for specifying embedding transformations in a graphical notation — are also part of PAGG, PROGRES, and DELTA.

The following example summarizes the discussion of possible interpretations of simple graph replacement rules:

Example 4.1 Translation of DeleteMan into a Structure Replacement Rule. The translation of the graph replacement rule DeleteMan of Figure 9 into a structure replacement rule in accordance with the informal discussion above has about the following form:

- (1) Left-hand side translation:

$$L := \{\text{node}(v1, \text{man}), \text{node}(v2, \text{woman}), \text{node}(v3, \text{man}), \\ \text{edge}(v2, \text{child}, v1), \text{edge}(v3, \text{child}, v1)\}$$
 plus optional extension for deletion of otherwise dangling edges with $S1, \dots$ being set identifiers:

$$\cup \{\text{edge}(v1, \text{child}, S1), \dots\}.$$
- (2) Right-hand side translation:

$$R := \{\text{node}(v2, \text{woman}), \text{node}(v3, \text{man})\}.$$
- (3) Dangling edge preventing (post-)condition translation:

$$AR := \{\forall x, e, y : \text{edge}(x, e, y) \rightarrow (\exists l : \text{node}(x, l) \wedge \exists l : \text{node}(y, l))\}.$$
- (4) Optional identification (pre-)condition:

$$AL := \{v1 \neq v3\}$$
 or/and optional induced subgraph (pre-)condition:

$$AL := \{\neg \exists e : \text{edge}(v2, e, v3) \wedge \dots\}$$
 or/and — in this case useless — isomorphic match (pre-)condition:

$$AL := \{v1 \neq v2, v1 \neq v3, v2 \neq v3\}. \quad \square$$

4.2 Embedding Rules and Path Expressions

Using the style of graph replacement introduced in the previous subsection we have no chance to define a rule which

- deletes a man node v with an unknown number of emanating child edges
- and creates for any deleted child edge from v to another node v' two new grandchild edges from the parents of v to v' .

This a scenario which requires *complex embedding transformation rules*. EXP is an early graph replacement approach which is famous for its powerful embedding rules. These rules have the following four components:

- A left-hand side node identifier, whose match in the host graph is potentially connected to an arbitrary number of unmatched context nodes.
- A so-called path expression describing a path through the host graph from the selected left-hand side node match to a relevant set of unmatched context nodes.
- A node of the right-hand side, whose copy in the host graph will be the source or target of a new embedding edge.
- A last part which determines the label and the direction of the new edge.

In the simplest case, a *path expression* of the form R_{label} or L_{label} requires the traversal of a single edge in or against its direction as for instance in

$$l_{\text{grandchild}} = (v2, v3; R_{\text{child}}(v1)) .$$

This embedding rule completes the rule DeleteMan and creates for any child edge from $v1$ to a context node v new grandchild edges from $v2$ and $v3$ to v . In the general case, path expressions are constructed by building concatenations, unions, iterations, etc. of simple path expressions. They may be used for defining *derived relationships* like ancestor or brother of Figure 2.

The reader may imagine that path expression-based embedding rules are a rather powerful concept, but are sometimes difficult to understand. Therefore, PAGG and DELTA rely on *graphical embedding rules*, whereas PROGRES offers both textual embedding rules like EXP and graphical embedding rules like DELTA. All graphical embedding rules have in common that they mark certain nodes (edges) in their graph replacement rules which are allowed to match an arbitrary number of nodes (edges) in the host graph. In PROGRES, these nodes are depicted as double dashed rectangles (cf. Figure 5 in Subsection 2.3 and Figure 10). The underlying formalism of structure replacement rules supports this concept by means of set identifiers:

Example 4.2 Translation of Embedding Rules.

The translation of the PAGG, PROGRES, and DELTA graph replacement rules of Figure 10 into an equivalent structure replacement rule is the same as in Example 4.1, except the fact that L and R are extended as follows:

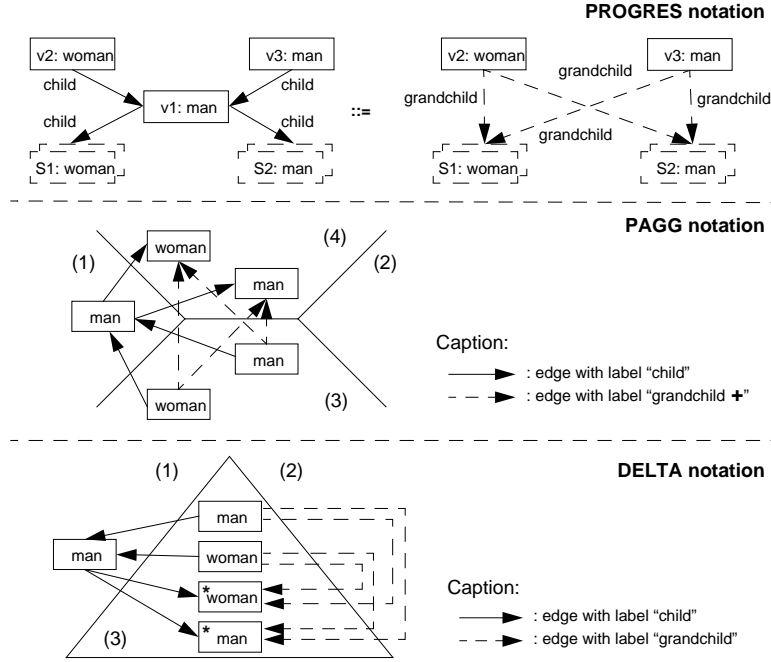


Figure 10: Notations for graph replacement rules with graphical embedding rules

- (1) $L' := L \cup \{\text{node}(S1, \text{woman}), \text{node}(S2, \text{man}), \text{edge}(v1, \text{child}, S1), \text{edge}(v1, \text{child}, S2)\}$.
- (2) $R' := R \cup \{\text{node}(S1, \text{woman}), \text{node}(S2, \text{man}), \text{edge}(v2, \text{grandchild}, S1), \text{edge}(v2, \text{grandchild}, S2), \text{edge}(v3, \text{grandchild}, S1), \text{edge}(v3, \text{grandchild}, S2)\}$.

$S1$ and $S2$ are node set identifiers which match zero or one or two or ... nodes in a host graph (cf. Def. 2.1). \square

The *PAGG notation* for the graph replacement rule in Figure 10 must be read as follows:

- (1) Nodes and edges in the left section of the X are deleted together with all incident edges during the application of such rewrite rule.
- (2) Nodes and edges in the right section of the X are created anew when the rewrite rule is applied.
- (3) Nodes and edges of the lower section of the X denote the preserved part of the host graph which has to be present but remains unmodified.

- (4) And nodes and edges of the upper section of the X denote graphical embedding rules. They are preserved and have in the general case not a single match, but a maximal number of matches in the host graph.

The meaning of *section crossing edges* should be obvious except those between the lower and the upper part of the X . They wear either the additional label “-” or “+”. The label “-” denotes those edges which have to be present before the application of the rule but are no longer present afterwards. The label “+” denotes those edges which need not be present before the application of the rule, but will be present afterwards.

DELTA *grammar rules* are rather similar to PAGG rewrite rules (cf. Figure 10). The area left of the Δ corresponds to the left section of the X , the area right of the Δ to the right section of the X , and the inner area of the Δ to the upper and the lower part of the X . Preserved nodes with multiple matches instead of single matches are identified by having an additional “*” label. The purpose of the region below the Δ will be explained within the next subsection.

4.3 Positive and Negative Application Conditions

It is quite often the case that a graph replacement step may only take place if additional nodes and edges are present or if a certain graph pattern is not present. Therefore, PROGRES, DELTA, and SPO know the concept of *positive and negative context conditions*. A very general version of these additional application conditions is suggested in [28] as an extension of the SPO approach. There, arbitrarily large context graphs may be attached to the left-hand side of a graph replacement rule, and their existence may be required or prohibited. DELTA has about the same concept of negative context graphs. They are depicted below the bottom line of the Δ . Positive context conditions correspond to the preserved part of the graph rewrite rule, the inner region of the Δ . In this way, SPO and DELTA are able to require the existence or absence of context subgraphs which have an *a priori known size*. The lower part of Figure 11 shows an example of a DELTA replacement rule which marries two previously unmarried persons. The preconditions require that the two nodes do not have a wife edge in between them. They require furthermore that the two nodes do not have any wife edge connections to other nodes.

The Figure 11 contains in its upper part the corresponding PROGRES definition with an additional restriction: The two nodes may not be ancestors of each other. This an example of a *negative context condition*, where the size of the prohibited subgraph — a path through a given host graph — is not fixed within the rule. Such a requirement is not expressible within SPO or

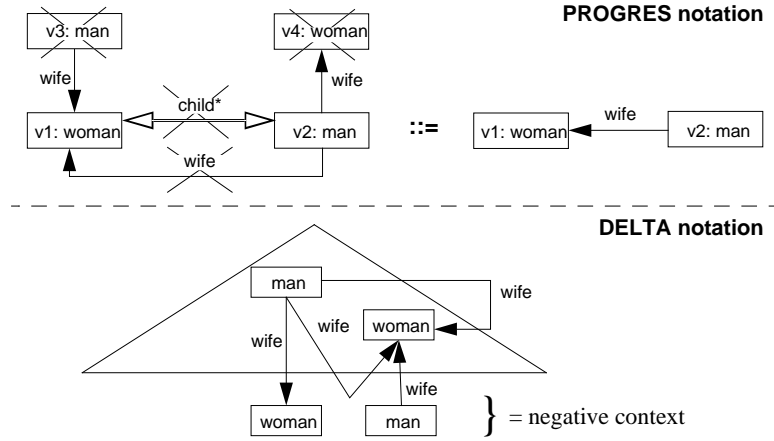


Figure 11: PROGRES and DELTA rules with negative context conditions

DELTA. PROGRES reuses the concept of path expressions — originally used for defining n -context embedding rules — to overcome this restriction. In such a way, control structures of path expressions for conditional branching, transitive closure, and conditional iteration are now available for defining *complex n -context application conditions*.

The translation of positive as well as negative application context conditions into preconditions of structure rewrite rules is rather obvious. Example 2.20 of Subsection 2.2 presents already the complete translation of a subgraph test with similar context conditions as in Figure 11. Furthermore, the translation of complex path expressions with transitive closure operators and the like is already part of Example 2.11 in Subsection 2.1.

4.4 Normal and Derived Attributes

Until now, we have discussed only those means of graph replacement rules which allow the manipulation of nodes and edges, i.e. of data which has an *important internal structure* from the specifiers point of view. But in many cases additional data is manipulated — like strings or numbers — which should be handled as *atomic items*. Therefore, all presented graph replacement approaches — except EXP — support the definition of *attributes*, which are attached to nodes and which have a single item or a set of atomic items as their values. In order to be able to write or read these attributes, textual extensions of graph replacement rules are offered. They are always more or less similar to

```

node class person;
  derived
  noOfAncestors = [ 1 + self.<-child-.instance of man.noOfAncestors | 0 ]
                  + [ 1 + self.<-child-.instance of woman.NoOfAncestors | 0 ];
end ;

```

Figure 12: PROGRES definition of a derived attribute

those of the language PROGRES, which were already used in Example 2.22 of Subsection 2.3. The graph replacement rule `Marry` presented there tests and modifies the `income` attribute of a matched `person` node.

In general, two different categories of attributes may be distinguished as they are sometimes distinguished within attribute tree grammars:

- *Intrinsic attributes*, like the above mentioned `income` attribute, represent extensionally defined properties of nodes in a graph. They change their value only through explicit assignments within graph replacement rules.
- *Derived attributes*, on the other hand, represent intentionally defined node properties. They are defined by means of directed equations over attributes of other nodes, which are in the n -context of their own node.

PROGRES supports both categories of attributes in their pure form. Normal attributes are tested and manipulated within graph replacement rules (cf. Example 2.22 and Figure 5 of Subsection 2.3). Derived attribute value definitions are part of a graph schema construction. Again path expressions are welcome for specifying rather *complex n -context references to attributes* at foreign nodes within directed equations. Figure 12 shows an example of such an attribute specification. The number of ancestors of a distinct person is the sum of the number of ancestors of her mother plus one (if existent) and the number of ancestors of her father plus one (if existent). In the case of nonexistence of the mother or the father the evaluation of the subexpression between `[... |` fails and the subexpression `0` between `|...]` is returned.

Example 4.3 Translation of Derived Attribute Definition.

The attribute equation of Figure 12 may be translated as follows into a formula, which is part of a structure schema definition (cf. Definition 2.9):

$$\begin{aligned}
& \forall x: \text{node}(x, \text{person}) \rightarrow \\
& \exists v: (\text{attr}(x, \text{noOfAncestors}, v) \\
& \quad \wedge (\exists v1, v2 : v = v1 + v2 \\
& \quad \quad \wedge (\exists x1: (\text{edge}(x1, \text{child}, x) \wedge \text{type}(x1, \text{man}) \wedge \text{attr}(x1, \text{noOfAncestors}, v1)) \\
& \quad \quad \quad \vee ((\neg \exists x1: \text{edge}(x1, \text{child}, x) \wedge \text{type}(x1, \text{man})) \wedge v1 = 0)) \\
& \quad \quad \wedge (\exists x2: (\text{edge}(x2, \text{child}, x) \wedge \text{type}(x2, \text{woman}) \wedge \text{attr}(x2, \text{noOfAncestors}, v2)) \\
& \quad \quad \quad \vee ((\neg \exists x2: \text{edge}(x2, \text{child}, x) \wedge \text{type}(x2, \text{woman})) \wedge v2 = 0))))). \quad \square
\end{aligned}$$

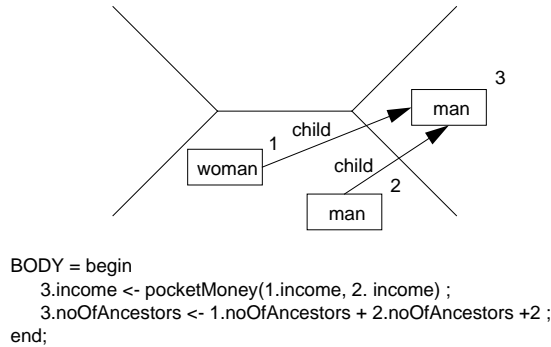


Figure 13: Attribute manipulating graph replacement rule in PAGG

Specifying derived attributes within graphs and keeping them consistent to each other is much more difficult if the underlying structure is an arbitrary directed graph and not a tree. The attempt to traverse an edge during the evaluation of a directed equation may fail due to the nonexistence of such an edge. On the other hand, it might be the case that a path expression within an attribute equation returns a set of nodes instead of a single node. Therefore, additional means are necessary for determining alternatives in the case of *partially defined* subexpression and for *aggregating sets* of attribute values to a single useful attribute value. For a detailed discussion of various *incremental attribute evaluation strategies* the reader is referred to [30,35,64]. Their presentation is outside the scope of this contribution.

The developers of PAGG and of another attribute graph grammar approach [67] adopted the idea of attribute tree grammars to *attach directed attribute equations to structure replacement rules* (productions) instead of defining them separately. As a consequence, derived attribute definitions and value assignments to extensionally defined attributes are handled by the same syntactic construct (cf. Figure 13), and we are not able to distinguish between

- (1) the case where an attribute, like `income`, gets once a value computed from other attribute values and preserves its value afterwards if these foreign attributes change values,
- (2) and the case where an attribute, like `noOfAncestors`, has to change its value as soon as referenced foreign attributes change their values.

Furthermore, attaching equations to rules has also the *drawback* that equations within different but somehow related rules — which insert, modify, and delete for instance a certain type of nodes — may become *inconsistent* to each other.

This is not a real problem as long as graph replacement systems are only used for generating graph languages, but tends to be a serious disadvantage as soon as graph replacement systems are used for specifying (abstract) data types. Nevertheless, the attachment of attribute equations to graph replacement rules has some advantages, too. It is no longer necessary to define a separate graph schema, the equations which reference nodes in rules (instead of using path expressions for this purpose) are easier to read, and the derivation history of a graph may influence the selection of active attribute equations.

Finally, we have to mention that EXP, DPO, and DELTA do not know the concept of derived attributes, whereas an SPO variant was defined very recently which supports derived data definition at least to a certain extent [32].

4.5 Summary

The following table summarizes the presentation of six more or less different graph replacement approaches within the previous subsections. Being only a rather condensed version of the preceding discussion it should be readable without any further explanations. During its construction we had to solve the *following problems*:

- The available input for DELTA was not sufficient for answering all questions and for verifying that DELTA belongs indeed to the family of algebraic graph replacement systems. Therefore, its column contains three question marks.
- The headlines SPO and DPO represent not only a single paper or proposal, but a whole family of related formalisms which evolved over the time. Therefore, it was very difficult to give definite “yes” or “no” answers in some cases.
- The classification of embedding rules is traditionally more fine-grained than the classification used in Subsection 4.2. Therefore, additional embedding rows were added for those readers who are familiar with graph replacement systems.

		Algorithmic Approaches			?	Algebraic Approaches			
		EXP [43, 44]	PAGG [26, 27]	PROGRES [61, 62]	$\Delta = \text{DELTA}$ [32]	DPO [17, 18]	SPO [18, 38]		
Graph Model	Node Label Hierarchy	—	in [24]	+	+	in [57]	—		
	Edge Label Hierarchy	—	—	—	+		—		
	Edge Identifiers	—	—	—	?	+	+		
	Hyperedges	—	—	—	—	+	+		
	Derived Edges = Path Expr.	+	—	+	—	—	—		
	“Normal” Node Attributes	—	+	+	special label	in [57]	in [37]		
	Derived Node Attributes	—	mixture	+	—	—	in [33]		
Separate Schema Definition		—	in [24]	+	—	in [15]			
Graph Replacement Rules	Matching	Homo-/Isomorphic	isomorphic	isomorphic	homo./iso.	homo./iso.	homomorph	homomorph	
		(Induced) Subgraph	induced sub.	subgraph	subgraph	subgraph	subgraph	subgraph	
		Multiple Matches in Host Graph (simultaneously)	(+) parallel or mixed rewr.	+	+	+	+	rule amal-gamation in [68]	rule amal-gamation in [30]
	Appl. Conditions	Explicit Cond.	Implicit Conditions	induced sub.	—	—	—	gluing cond.	—
			Attribute Cond.	—	+	+	+	in [57]	in [37]
			Positive Context	—	any graph \rightarrow n-C. (fix)	nodes/paths \rightarrow n-Context	any graph \rightarrow n-C. (fix)	any graph \rightarrow n-C. (fix)	any graph \rightarrow n-C. (fix)
			Negative Context	—	—	nodes/paths \rightarrow n-Context	any graph ? \rightarrow n-C. (fix)	—	graph in [29] \rightarrow n-C. (fix)
	Delete	Del./Preserve Conflict	impossible	impossible	forbidden	forbidden ?	forbidden	delete	
		Handling of Dangling Context Edges	automatic deletion	automatic deletion	automatic deletion	automatic deletion	forbidden gluing cond.	automatic deletion	
	Embedding Transformation	Textual/Graphical	text (paths)	graphical	text/graphic	graphical	(simulation with amal-gamation possible)	(simulation with amal-gamation possible)	
		Referenced Context	n-Context	n-C. (fix)	n-Context	n-C.(fix)			
		Change Orientation	+	+	+	+			
		Change Label	+	+	+	+			
		Test Context Label	+	+	+	+			
		Test Context Edges	+	+	+	+			
With Preserved Nodes		—	+	+	+				
Different Treatment of Ident. Labeled Nodes	+	+	+	+					

Table 1: Comparison of algorithmic and algebraic graph replacement approaches

5 Programmed Graph Replacement Systems — An Overview

Section 3 introduced the basic means for defining and comparing various forms of *sequential programmed graph replacement systems*. It is the topic of this section to review previously made proposals for programmed grammars or — more specific — programmed graph grammars or replacement systems. The review is divided into three subsections:

- Subsection 5.1 introduces a number of programmed grammar formalisms which preserve the declarative as well rule-oriented character of replacement systems. There exists still a *single meta algorithm* which applies rules as long as appropriate and which takes additional information about rule preferences or required application sequences into account.
- Subsection 5.2 presents then a number of related programmed graph grammar approaches which belong to the application-oriented branch of algorithmic graph grammars. These approaches have in common that they adapt *control structures* of imperative programming languages for the purpose of controlling graph replacement rules.
- Finally, Subsection 5.3 deals with all those programmed graph grammars which belong to the family of algorithmic graph grammar approaches, too, but use *control flow diagrams* instead of textually written control structures for regulating graph replacement processes.

We will use a single example of a complex person database transformation to discuss the advantages and disadvantages of all presented approaches. This is an extended version of the transaction `Marry*` of Example 3.1. It assumes the existence of already introduced basic rules `Marry` and `MarkUnmarried` (cf. Example 3.1) as well as the new rule `DeleteUnmarried`. The latter one fails if all person nodes are married and deletes a nondeterministically selected unmarried person node otherwise.

Example 5.1 Complex Database Transformation.

```
Marry* = (((def(MarkUnmarried); Marry); Marry*) || undef(MarkUnmarried)).  
Main   = (Marry* || (undef(Marry*); (DeleteUnmarried; Main))). □
```

The transaction `Main` tries first to marry all singles in the database and terminates successfully if possible. Otherwise, the subtransaction `Marry*` fails and `undef(Marry*)` succeeds. As a consequence, `DeleteUnmarried` removes one person node from the database and `Main` calls itself again. The whole process stops as soon as all singles are either married or deleted.

5.1 Declarative Rule Regulation Mechanisms

This subsection presents three types of programmed graph grammars which preserve the *rule-oriented and declarative character* of rule-based systems. All of them offer only (very) limited support for determining the order of rule applications and have still the main *match and replace* loop of pure rule based systems. The main loop terminates if and only if no more rules are applicable. The result is “per definition” an element of the specified language as long as terminals and nonterminals are not distinguished.

Having *no user defined termination conditions* at hand and almost no means to distinguish between failing and successful derivation sequences, it would be very difficult to simulate the behavior of the transactions **Main** and **Marry***. Additional nonterminals, application conditions, and even new rules would be necessary to guarantee for instance that **DeleteUnmarried** does not fire before all possible derivation sequences for **Marry*** failed. Therefore, we will not define our running example with the following formalisms:

Definition 5.2 Graph Grammars with Rule Priorities.

A **graph grammar with rule priorities** has an axiom plus a set of graph replacement rules P together with a partial order “ $<$ ” such that:

$$P := \{p_{1,1}, \dots, p_{1,k(1)}\} > \dots > \{p_{n,1}, \dots, p_{n,k(1,n)}\}$$

with $n \in N$ and $k : N \rightarrow N$.ⁱ

The following transaction defines the semantics of rule priorities:

$$t = ((t'; t) \parallel \underline{\text{skip}})$$

with

$$t' = (t_1 \parallel (\underline{\text{undef}}(t_1); t_2) \parallel \dots \parallel ((\underline{\text{undef}}(t_1) \parallel \dots \parallel \underline{\text{undef}}(t_{n-1})); t_n))$$

$$t_i = (p_{i,1} \parallel \dots \parallel p_{i,k(i)}) \text{ for } i := 1, \dots, n. \quad \square$$

This *definition of rule priorities* is just an *approximation* of their usual semantics. According to their definition over here, the execution of a lower priority rule is blocked as soon as a higher priority rule is executable, even in the case where both rules modify disjoint parts of a given graph. Unfortunately, a more liberal definition of rule priorities is beyond the capabilities of BCF expressions. This is a principle problem of our approach, since enlarging an already nonempty set of matches within a graph for a rule p may reduce the set of allowed matches of another graph replacement rule p' with lower priority. This is a kind of nonmonotonic behavior, we are not able to deal with.

A quite different rule regulation mechanism is studied in [13], so-called *matrix (graph) grammars*. They are simply sets of sequences of rules and have the following semantics:

ⁱ $N := \{0, 1, \dots\}$ is the set of all natural numbers including zero.

Definition 5.3 Matrix Graph Grammars.

A **matrix graph grammar** has an axiom plus a set of graph replacement rule sequences of the form

$$\{(p_{1,1}, \dots, p_{1,k(1)}), \dots, (p_{n,1}, \dots, p_{n,k(n)})\} \text{ with } n \in N \text{ and } k : N \rightarrow N.$$

The following transaction defines the semantics of such a set of sequences:

$$t = ((t'; t) \parallel \text{skip})$$

with

$$t' = (t_1 \parallel \dots \parallel t_n)$$

$$t_i = ((p_{i,1} \parallel \text{undef}(p_{i,1})); \dots; (p_{i,k(i)} \parallel \text{undef}(p_{i,k(i)})) \text{ for } i := 1, \dots, n,$$

i.e. rule sequences are selected and executed similar to simple rules of a grammar. Rules in a sequence are applied if possible and skipped otherwise. \square

Another formalism studied in [13] are so-called *programmed (graph) grammars* which are at the borderline between purely declarative regulation mechanisms and programmed approaches in the sense of the following Subsection 5.2. They allow for the definition of atomic sequences of rule applications and support branching in the flow of control depending on success or failure of single rules. But without any support for functional abstraction they do not allow branching in the flow of control depending on success or failure of complex graph replacement processes.

Definition 5.4 Programmed Graph Grammars.

A **programmed graph grammar** over a set of graph replacement rules P in the sense of [13] has an axiom plus a set of triples (p_i, T_i, F_i) for $i := 1, \dots, n$ with components being defined as follows:

- (1) $p_i \in P$ is the rule which has to be applied.
- (2) $T_i := \{p_{i,1}, \dots, p_{i,k(i)}\} \subseteq P$ is the set of rules which may be applied after a successful replacement step with p_i .
- (3) $F_i := \{p'_{i,1}, \dots, p'_{i,k'(i)}\} \subseteq P$ is the set of rules which may be applied if the application of p_i fails.

The following transaction defines the semantics of this type of grammar:

$$t = (t_1 \parallel \dots \parallel t_n)$$

with

$$t_i = (p_i; (t_{i,1} \parallel \dots \parallel t_{i,k(i)})) \parallel (\text{undef}(p_i); (t'_{i,1} \parallel \dots \parallel t'_{i,k'(i)}))$$

where

t_i is the translation of $(p_i, \{p_{i,1}, \dots, p_{i,k(i)}\}, \{p'_{i,1}, \dots, p'_{i,k'(i)}\})$,

$t_{i,j}$ is the translation of $(p_{i,j}, \dots)$ for $j := 1, \dots, k(i)$, and

$t'_{i,j}$ is the translation of $(p'_{i,j}, \dots)$ for $j := 1, \dots, k'(i)$. \square

5.2 Programming with Imperative Control Structures

Previously introduced regulation mechanisms were not adequate for defining complex graph transformation processes as those of our running Example 5.1. Therefore, new mechanisms had to be found, when people started to use graph grammars for specifying complex data structures of software engineering environments [19] or CAD systems [24]. In these cases it was quite natural to use *control structures* of *imperative programming languages* and to adapt them for the new purpose of regulating graph replacement processes.

At the beginning, formal definitions of this kind of programmed graph grammars were rather vague. Furthermore, they didn't make any attempts to resolve the conflict between *nondeterministically* working graph replacement rules and *deterministically* working control structures. Typical representatives of this category may be found in [21] and [36]. They are the direct predecessors of a new generation of programmed graph grammars which will be reviewed within this subsection.

Let us start with the definition of a simplified version of control structures within the language PROGRES (cf. [64,69]).

Definition 5.5 Control Structures of PROGRES.

The control structures of **PROGRES** have about the following form and semantics^j:

- (1) fail $\hat{=}$ undef(skip),
the always failing command.
- (2) do a or b end $\hat{=}$ $(a \parallel b)$,
the nondeterministic branch statement which executes either a or b .
- (3) if def a then b else c end $\hat{=}$ $((\text{def}(a); b) \parallel (\text{undef}(a); c))$,
the deterministic branch statement which applies b if a would be applicable (testing applicability of a does not cause any graph modifications) and c otherwise.
- (4) atom a; b end $\hat{=}$ $(a; b)$,
the deterministic and atomic sequence which fails whenever a or b fails.
- (5) do a and b end $\hat{=}$ $((a; b) \parallel (b; a))$,
the nondeterministic sequence operator which executes a and b in arbitrary order.
- (6) while def a do b end $\hat{=}$ call of t with $t = ((\text{def}(a); (b; t)) \parallel \text{undef}(a))$,
the conditional iteration which executes b as long as a would be executable. □

^jThe syntax is simplified for readability reasons and additionally available abbreviations for often occurring control flow patterns like “if def a then a else b end” etc. are not discussed.

Example 5.6 The Transformation Main in PROGRES.

Using the control structure of Definition 5.5 we are able to produce a quite readable version of our running Example 5.1.

Marry* = while def MarkUnmarried do Marry end.
 Main = if def Marry* then Marry* else atom DeleteUnmarried; Main end end. \square

Another definition of *programmed attributed graph grammars* (PAGG) with imperative control structures may be found in [24,26]:

Definition 5.7 Control Structures of PAGG.

Control structures in **PAGG** have about the following form and semantics^k:

- (1) begin $a; b$ end $\hat{=}$ $((\text{def}(a) \parallel \text{def}(b)); (a \parallel \text{undef}(a)); (b \parallel \text{undef}(b)))$.
- (2) try a else b end $\hat{=}$ $(a \parallel (\text{undef}(a); b))$.
- (3) repeat a end $\hat{=}$ call of t with $t = (a; (t \parallel \text{undef}(a)))$. \square

These control structures *compromise* the required *boolean nature* and *atomic character* of graph transactions in favor of an efficiently working implementation. They have to be read as follows:

- The construct begin $a; b$ end requires sequential application of a followed by b . Its definition is rather complicated since its execution fails only if both a and b fail. Otherwise, the failing subexpression is skipped and the execution process continues.
- The construct try a else b end has the same semantics as the PROGRES control structure if a then a else b end. It may even be used with more than two arguments in the general case.
- The construct repeat a end applies a as long as possible and succeeds if a is applied at least once. It may also be used with more than one argument between repeat and end, but this is just a shorthand for a begin/end sequence within repeat/end.

Using the nonatomic sequence statement and offering no means which are equivalent to the previously introduced BCF operators def and undef (cf. Def. 3.6) reduces the expressiveness of PAGG significantly, but has the advantage that an implementation has *no needs for backtracking* and undoing already performed graph modifications. Therefore, we are not able to provide the reader with a semantically equivalent PAGG version of transformation

^kThe original definition uses sapp instead of begin, capp instead of try and wapp instead of repeat.

Main. Nevertheless, we will try to approximate the intended meaning as far as possible in order to highlight the deficiencies of PAGG control structures:

Example 5.8 The Transformation Main in PAGG.

```
Marry* = begin repeat Marry end; < fail if unmarried person exists > end.  
Main   = repeat try Marry* else DeleteUnmarried end end. □
```

The problem with the example above is that we have no means to test whether a given graph replacement rule would be applicable without actually executing it. And even replacing the needed test < fail if... > by a sequence like begin MarkUnmarried; Unmark end is useless, since unapplicable rewrite rules do not cause the failure of a sequence, but are simply skipped.

Finally, we should emphasise that the discussed deficiencies of PAGG control structure are not relevant as long as specified graph transformation processes are *deterministic* or *confluent*. In this case, PAGG control structures are not only efficiently executable, but meet exactly the needs of their users.

Another recently developed *graph grammar programming approach* [14] has about the same properties as PAGG and is, therefore, not discussed in further detail. It offers merely a combination of the already presented or of PROGRES, its if then else with a restriction to boolean conditions, and finally the nonatomic begin/end of PAGG.

5.3 Programming with Control Flow Graphs

Previous subsections presented typical exponents of various programmed graph replacement approaches. In all cases denotational semantics definitions were sketched by translating their control structures into BCF expressions with a well-defined fixpoint semantics definition. But all discussed approaches — except BCF expressions themselves and their more readable form in the language PROGRES — did not fulfill the list of requirements in Subsection 3.1. And even BCF expressions with their low level text representation compromise the *visual nature* of programming with graph replacement rules.

An obvious solution for this problem is to replace text-oriented control structures by some kind of *control flow graphs*. But almost all variants of control flow graphs do not fulfill all needed properties of control structures, too:

- Control flow graphs which are not allowed to call each other, i.e. do not properly support functional abstraction, are defined in [9,10].
- In [43] control flow graphs are introduced, where a node may be a call to another control flow graph, but where branching depending on success or failure of subgraph calls is not supported.

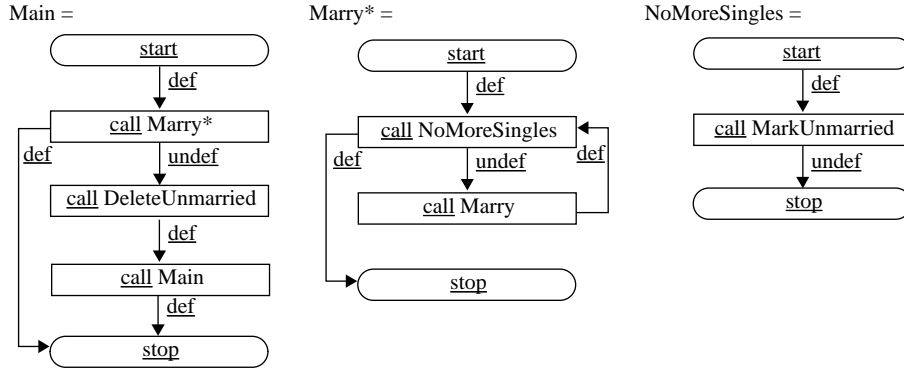


Figure 14: The control flow graph version of transformation Main

- And [64] presents a variant of control flow graphs without the above mentioned deficiencies, but also without a nonoperational semantics definition.

In the sequel, we will show how BCF expressions may be used to define a fixpoint semantics for the control flow graphs of [64]. These control flow graphs have about the same properties as previously defined transactions (boolean character etc.). Furthermore, they have the same expressive power as BCF expressions with the strict def^∞ operator version and without the intersection operator $\&$. Figure 14 contains an example of a control flow graph with two additional subgraphs. Their main properties are:

- Any vertex in a subgraph should be reachable from its unique start vertex and should have a path to its unique stop vertex.
- The execution of a (sub-)graph begins at its start vertex and ends at its stop vertex.
- Normal vertices in the graph have a basic action as inscription, which is either nop or the call of a simple rewrite rule or (the start vertex of) another control flow subgraph.
- The execution of a vertex either succeeds or fails; execution of start, stop, and nop vertices always succeeds without any effects.
- After a *successful* execution of a vertex we have to follow one of its outgoing def edges; multiple def edges reflect nondeterministic computations and absence of def edges partially defined computations.
- After a *failing* execution of a vertex we have to follow one of its outgoing undef edges; multiple undef edges reflect again nondeterministic computations and absence of undef edges again partially defined computations.

For a more detailed explanation of this kind of control flow graphs, including an operational semantics definition, the reader is referred to [64]. A *denotational semantics definition* may be provided by translating a control flow graph into an equivalent set of transactions (BCF expressions). Any vertex of the control flow graph will be translated into a transaction with calls to other transactions reflecting its outgoing def and undef edges:

Definition 5.9 Graph Replacement Systems with Control Flow Graphs.

The semantics of a set of **control flow (sub-)graphs** over a set of graph replacement rules P and a set of control flow vertices V is defined as follows: Any vertex v of the control flow graphs with inscription “call a ” or “start” or “nop” with outgoing “def” edges to vertices v_1, \dots, v_k , and outgoing “undef” edges to vertices $v'_1, \dots, v'_{k'}$ is translated into:

$$v = ((x; (v_1 \parallel \dots \parallel v_k)) \parallel (\text{undef}(x); (v'_1 \parallel \dots \parallel v'_{k'}))), \text{ if } k, k' > 0,$$

and into

$$v = (x; (v_1 \parallel \dots \parallel v_k)), \quad \text{if } k > 0, k' = 0$$

$$v = (\text{undef}(x); (v'_1 \parallel \dots \parallel v'_{k'})), \quad \text{if } k = 0, k' > 0$$

and into

$$v = \text{fail} = \text{undef}(\text{skip}), \quad \text{if } k = 0, k' = 0$$

where

$$x = a, \quad \text{if inscription of } v \text{ is “call } a\text{” and } a \in P \cup V$$

$$x = \text{skip}, \quad \text{otherwise.}$$

Finally, any vertex n with inscription “stop” is translated into the transaction

$$v = \text{skip},$$

thereby terminating the execution of its control flow subgraph successfully. \square

Please note that this definition of control flow graphs is almost identical to the Definition 5.4 of programmed graph grammars. The essential difference is that control flow graphs know the concept of functional abstraction. Thereby, calls to complex subdiagrams may be treated in the same way as calls to simple rules. In this way, arbitrary forms of *recursion* are supported and *complex conditions* may be defined which cause *failure* or successful *termination* of derivation (sub-)processes.

Proposition 5.10 Equivalence of BCF Expressions and Control Flow Graphs.

Any set of transactions with BCF expressions as their bodies — without the intersection operator $\&$ and with a strict def^∞ operator version instead of the original def version (cf. Definition 3.6 and Definition 3.7) — may be translated into an equivalent set of control flow graphs according to Definition 5.9 and vice versa.

	Sequ. Atomic Control Flow	Nondeterm. Control Flow	Success & Failure Test	Recursive Calls	Paradigm & Notation
Simple Rule Priorities (here)	no	no	no	no	declarative & text-oriented
Matrix Grammars [13]	atomic sequence	no	no	no	decl./imperative & text-oriented
Programmed Grammars [13]	nonatomic sequence	no	rules only	tail recursion	decl./imperative & text-oriented
Prog. Attributed Graph Grammars [25,26]	nonatomic sequence	no	rules only	yes	imperative & text-oriented
Prog. Graph Rewriting Systems [14]	nonatomic sequence	yes	no	yes	imperative & text-oriented
PROGRES [64] & BCF Expressions [66]	yes	yes	yes	yes	imperative & text-oriented
Prog. Deriv. of Rel. Structures [42]	yes	yes	no	yes	parallel imper. & text-oriented
Control Flow Diagrams in [9,10]	yes	yes	no	no	imperative & graphical
Control Flow Diagrams in [43]	yes	yes	rules only	yes	imperative & graphical
New Control Flow Diagrams [66]	yes	yes	yes	yes	imperative & graphical

Table 2: A comparison of various rule-regulation mechanisms

Proof:

The translation from control flow graphs into BCF expressions is already part of their Definition 5.9. The other direction follows also directly from Definition 5.9. Rule sequences, nondeterministic branching, the operator skip, and calls to subprograms are directly supported. And undef(*a*);... may be simulated by a vertex with inscription *a*, no outgoing def edge and a single outgoing undef edge to another vertex which represents the following computation step within the expression undef(*a*);... Finally, an expression of the form def[∞](*a*) is equivalent to undef(undef(*a*)) (cf. Definition 3.7). □

Using the construction of the proof above, we are now able to translate Example 5.1 into a control flow diagram and the BCF expressions of the control flow diagrams of Figure 14 into BCF expressions. Please note that the results of these translation processes are always more complex structured than their inputs. Therefore, Example 5.1 and Figure 14 are translations of each other (in the sense of Proposition 5.10) followed by a number of semantic preserving simplification steps (cf. def 3.6 and 5.9).

5.4 Summary

Within this section various approaches for sequential *programming with graph replacement rules* were defined and compared with each other by translating them into equivalent BCF expressions. The results of our discussion are summarized in Table 2 with column headlines referring to required rule programming properties in Subsection 3.1. This table shows that all proposed regulation mechanisms have their deficiencies. All of them — except PROGRES (Def. 5.5) and control flow graphs (Def. 5.9) — have difficulties with guaranteeing properties of simple graph replacement steps for complex transformation processes, too. Even worse, not a single approach is existent which supports parallel programming and testing success or failure of complex subtransformations at the same time. It is subject to future work to find out whether it is possible to combine the trace-based fixpoint semantics for parallel graph replacement systems in [43] with the fixpoint semantics approach over here for the definition of the operators def and undef (see also Subsection 3.5).

Acknowledgments

The author is most grateful to Manfred Nagl, Gregor Engels, Claus Lewerentz, Bernhard Westfechtel, Andreas Winter, Albert Zündorf, and other (former) members of the IPSEN project for many stimulating discussions about pro-

grammed graph replacement systems and their invaluable efforts to pave the way for the work presented here. Furthermore, thanks to the two anonymous referees of this chapter for their helpful comments on a previous version, to Andre Speulmanns for producing a first version of Table 1 (comparison of graph replacement approaches), and to Alexander Bell for his excellent Framemaker to L^AT_EX compiling job.

References

1. V. Claus, H. Ehrig, G. Rozenberg (eds.): *Proc. Int. Workshop on Graph Grammars and Their Application to Computer Science and Biology*, LNCS 73, Springer Verlag (1979)
2. H. Ehrig, M. Nagl, G. Rozenberg (eds.): *Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 153, Springer Verlag (1983)
3. H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (eds.): *Proc. 3rd Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 291, Springer Verlag (1987)
4. H. Ehrig, H.-J. Kreowski, G. Rozenberg (eds.): *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Springer Verlag (1991)
5. G. Ausiello, P. Atzeni (eds.): *Proc. Int. Conf. on Database Theory*, LNCS 243, Springer Verlag (1986)
6. D. Blostein, H. Fahmy, A. Grbavec: *Practical Use of Graph Rewriting*, TR No. 95-373, Computing and Information Science, Queen's University, Canada (1995)
7. P. Boehm, H.-R. Fonio, A. Habel: *Amalgamation of Graph Transformations with Applications to Synchronization*, in: H. Ehrig, C. Floyd, M. Nivat, J. Thatcher (eds.): *Mathematical Foundations of Software Development*, LNCS 185, Springer Verlag (1985), 267-283
8. F. Bry, R. Manthey, B. Martens: *Integrity Verification in Knowledge Bases*, in: [71], 114-139
9. H. Bunke: *Sequentielle und parallele programmierte Graphgrammatiken*, Dissertation, TR No. IMMD-7-7, Universität Erlangen, Germany (1974)
10. H. Bunke: *Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation*, IEEE Pattern Analysis and Machine Intelligence, Vol. 4, No. 6, IEEE Computer Society Press (1982), 574-582
11. B. Courcelle: *Graphs as Relational Structures: An Algebraic and Logical Approach*, in: [4], 238-252

12. E.W. Dijkstra: *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, in: Communications of the ACM, No. 18, acm Press (1975), 453-457
13. J. Dassow, G. Paun: *Regulated Rewriting in Formal Language Theory*, EATCS 18, Springer Verlag (1989)
14. H. Dörr: *Efficient Graph Rewriting and its Implementation*, Ph.D. Thesis, FU Berlin, LNCS 922, Springer Verlag (1995)
15. A. Corradini, R. Heckel: *A Compositional Approach to Structuring and Refinement of Typed Graph Grammars*, appears in: A. Corradini, U. Montanari (eds.): Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, Electronic Notes in Theoretical Computer Science (ENTCS), Amsterdam: Elsevier Science Publ. (1995), 167-176
16. H. Ehrig, A. Habel, H.-J. Kreowski, F. Parisi-Presicce: *From Graph Grammars to High-Level Replacement Systems*, in: [4], 269-291
17. H. Ehrig: *Introduction to the Algebraic Theory of Graph Grammars (a Survey)*, in: [1], 1-69
18. H. Ehrig, M. Korff, M. Löwe: *Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts*, TR No. 90/21, TU Berlin, Germany (1990)
19. G. Engels, C. Lewerentz, M. Nagl, Schäfer, A. Schürr: *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
20. G. Engels, C. Lewerentz, W. Schäfer: *Graph Grammar Engineering: A Software Specification Method*, in: [3], 186-201
21. G. Engels: *Graphen als zentrale Datenstrukturen in einer Software-Entwicklungsumgebung*, Dissertation, Universität Osnabrück, VDI Verlag (1986)
22. H. Fahmy, B. Blostein: *A Graph Grammar Programming Style for Recognition of Music Notation*, Machine Vision and Applications, Vol. 6, No. 2 (1993), 83-99
23. H. Gallaire: *Logic and Data Bases*, Plenum Press (1978)
24. H. Göttler, J. Günther, G. Nieskens: *Use Graph Grammars to Design CAD-Systems*, in [4], 396-410
25. H. Göttler: *Attribute Graph Grammars for Graphics*, in: [2], 130-142
26. H. Göttler: *Graphgrammatiken in der Softwaretechnik*, IFB 178, Springer Verlag (1988)
27. P. Van Hentenryck: *Constraint Satisfaction in Logic Programming*, MIT Press (1989)

28. A. Habel, R. Heckel, G. Taentzer: *Graph Grammars with Negative Application Conditions*, appears in: [55]
29. R. Heckel, J. Müller, G. Taentzer, A. Wagner: *Attributed Graph Transformations with Controlled Application of Rules*, appears in: [55]
30. S.E. Hudson: *Incremental Attribute Evaluation: an Algorithm for Lazy Evaluation in Graphs*, TR No. 87-20, University of Arizona (1987)
31. S. Kaplan, J. Loyall, S. Goering: *Specifying Concurrent Languages and Systems with Δ -Grammars*, in: [4], 475-489
32. M. Korff: *Algebraic Transformation of Equationally Defined Graph Structures*, TR No. 92/32, TU Berlin, Germany (1992)
33. H.-J. Kreowski, G. Rozenberg: *On Structured Graph Grammars: Part I and II*, TR No. 3/88, University of Bremen, FB Mathematik/Informatik, Germany (1988)
34. H.-J. Kreowski, *Graph Grammars for Software Specification and Programming: An Eulogy in Praise of Grace*, in: G. V. Feruglio, F. R. Llop-part (eds.), Proc. Colloquium on Graph Transformation and its Application in Computer Science, Technical Report, B-19, Universitat de les Illes Balears, 55-62
35. N. Kiesel, A. Schürr, B. Westfechtel: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*, Information Systems, Vol. 20, No. 1, Pergamon Press (1995), 21-51
36. C. Lewerentz: *Interaktives Entwerfen großer Programmsysteme, Konzepte und Werkzeuge*, Dissertation, RWTH Aachen, IFB 194, Springer Verlag (1988)
37. M. Löwe, M. Korff, A. Wagner: *An Algebraic Framework for the Transformation of Attributed Graphs*, in: [68], 185-199
38. M. Löwe: *Algebraic Approach to Graph Transformation Based on Single Pushout Derivations*, Ph.D. Thesis, TR No. 90/5, Fachbereich Informatik, TU Berlin, Germany (1990)
39. Z. Manna: *Mathematical Theory of Computation*, McGraw-Hill, USA (1974)
40. J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarkis: *Telos: a Language for Representing Knowledge about Information Systems*, ACM Transactions on Information Systems, Vol. 8, No. 4, acm Press (1990), 325-362
41. J. Minker: *Perspectives in Deductive Databases*, in: The Journal of Logic Programming, Elsevier Science Publ. (1988), 33-60
42. A. Maggiolo-Schettini, J. Winkowski: *Programmed Derivations of Relational Structures*, in: [4], 582-598
43. M. Nagl: *Graphgrammatiken*, Vieweg Verlag (1979)

44. M. Nagl: *A Tutorial and Bibliographical Survey on Graph Grammars*, in: [1], 70-126
45. M. Nagl: *Graph Technology Applied to a Software Project*, in: [54], 303-322
46. M. Nagl: *Characterization of the IPSEN Project*, in: [72], 196-201
47. Sh.A. Naqvi: *Some Extensions to the Closed World Assumption in Databases*, in: [5], 341-348
48. Sh.A. Naqvi: *Negation as Failure for First-Order Queries*, in: Proc. 5th SIGACT-SIGMOD Symp. on Principles of Database Systems, acm Press (1986), 114-122
49. G. Nelson: *A Generalization of Dijkstra's Calculus*, in: ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, acm Press (1989), 517-561
50. H. Noltemeier (ed.): *Graphen, Algorithmen, Datenstrukturen*, Proc. 2nd Workshop on Graphtheoretic Concepts in Computer Science (WG '76), Applied Computer Science 4, Hanser Verlag (1976)
51. M. Nagl, A. Schürr: *A Specification Environment for Graph Grammars*, in: [4], 599-609
52. Sh.A. Naqvi, Sh. Tsur: *Data and Knowledge Bases*, IEEE Computer Society Press (1989)
53. J.L. Pfaltz, A. Rosenfeld: *Web Grammars*, Proc. Int. Joint Conf. Artificial Intelligence, Washington (1969), 609-619
54. G. Rozenberg, A. Salomaa (eds.): *The Book of L*, Springer Verlag (1986)
55. G. Rozenberg (ed.): *Special Issue on Graph Transformation Systems*, Fundamenta Informaticae Vol.XXVII, No.1/2, IOS Press (1995)
56. J. Rekers, A. Schürr: *A Parsing Algorithm for Context-Sensitive Graph Grammars*, TR No. 95-05, University of Leiden, Netherlands (1995)
57. H. Rybinski: *On First-Order-Logic Databases*, in: ACM Transaction on Database Systems, Vol. 12, No.3, acm Press (1987), 325-349
58. G. Schied: *Über Graphgrammatiken: Eine Spezifikationsmethode für Programmiersprachen und verteilte Regelsysteme*, Dissertation, TR No. IMMD-25-2, Universität Erlangen (1992)
59. G. Schmidt, R. Berghammer (eds.): *Proc. Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '91)*, LNCS 570, Springer Verlag(1991)
60. H.J. Schneider: *Chomsky-Systeme für partielle Ordnungen*, TR No. IMMD-3-3, Universität Erlangen, Germany (1970)
61. H.J. Schneider, H. Ehrig (eds.): *Proc. Int. Workshop on Graph Transformations in Computer Science*, LNCS 776, Springer Verlag (1994)

62. H.J. Schneider: *Conceptual Database Descriptions Using Graph Grammars*, in: [50], 77-98
63. A. Schürr: *PROGRES: A VHL-Language Based on Graph Grammars*, in: [4], 641-659
64. A. Schürr: *Operationales Spezifizieren mit Programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungen und Werkzeuge*, Dissertation, RWTH Aachen, Deutscher Universitätsverlag (1991)
65. A. Schürr: *Logic Based Structure Rewriting Systems*, in: [61], 341-357
66. A. Schürr: *Logic Based Programmed Structure Rewriting Systems*, appears in: [55]
67. A. Schütte: *Spezifikation und Generierung von Übersetzern für Graphsprachen durch attributierte Graphgrammatiken*, Dissertation, EWH Koblenz, EXpress Edition (1987)
68. M.R. Sleep, M.J. Plasmeijer, M.C. van Eekelen (eds.): *Term Graph Rewriting: Theory and Practice*, John Wiley & Sons Ltd (1993)
69. A. Schürr, A. Zündorf: *Non-Deterministic Control Structures for Graph Rewriting Systems*, in: [59], 48-62
70. G. Taentzer, M. Beyer: *Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System*, in: [61], 380-394
71. A. Voronkov (ed.): *Logic Programming*, LNCS 592, Springer Verlag (1991)
72. H. Weber (eds.): *Proc. of the Int. Conf. on Systems Development and Factories*, Pittmann Press (1990)
73. B. Westfechtel: *Revisionskontrolle in einer integrierten Softwareentwicklungs-Umgebung*, Dissertation, RWTH Aachen, Informatik-Fachberichte 280, Springer Verlag (1991)
74. A. Zündorf: *Eine Entwicklungsumgebung für PROGRAMMIERTE GRAPH-ERSETZUNGSSYSTEME*, Dissertation, RWTH Aachen, Deutscher Universitätsverlag (1996)