Aachener Informatik-Berichte AIB 94-12

Specification of Graph Translators with Triple Graph Grammars

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

Abstract. *Data integration is a key issue for any integrated set of software tools where each tool has its own data structures (at least on the conceptual level), but where we have many interdependencies between these private data structures. A typical CASE environment, for instance, offers tools for the manipulation of requirements and software design documents and provides more or less sophisticated assistance for keeping these documents in a consistent state. Up to now almost all of these data consistency observing or preserving integration tools are hand-crafted due to the lack of generic implementation frameworks and the absence of adequate specification formalisms. Triple graph grammars, a proper superset of pair grammars, are intended to fill this gap and to support the specification of interdependencies between graph-like data structures on a very high level. Furthermore, they form a solid fundament of a new machinery for the production of batch-oriented as well as incrementally working data integration tools.*

1. Introduction

Graphs play an important role within many application areas of computer science, as e.g. in the form of data flow or control flow graphs in compiler construction, structured analysis and entity relationship diagrams in software engineering, or hypertexts in office automation. Furthermore, *rule-based systems* have proven to be well-suited for the description of complex transformation or inference processes on complex data structures.

Although graphs and rule-based systems are quite popular, their combination in the form of *graph rewriting systems or graph grammars* was more or less unknown for a very long time. Nowadays the situation is gradually improving with the appearance of graph rewriting system implementations like PAGG [6], GraphED [8], AGG [10], and PROGRES [16]. Especially the latter one has quite successfully been used within the project IPSEN for the development of an Integrated Project Support ENvironment [4, 15, 22], and within the project SUKITS for the development of an (a posteriori) integrated CIM environment [5].

Nevertheless, graph rewriting systems are usually restricted to the specification of processes which perform in-place modifications and transform one instance of a class of graphs into another instance of the same class. Therefore, they are not well-suited for the specification of compilers and integration or traceability tools which

- either take a complex data structure (source graph) as input and translate it into a new, separate data structure (target graph),
- or check consistency between different data structures,
- or propagate small changes of one data structure as incremental updates into another related data structure.

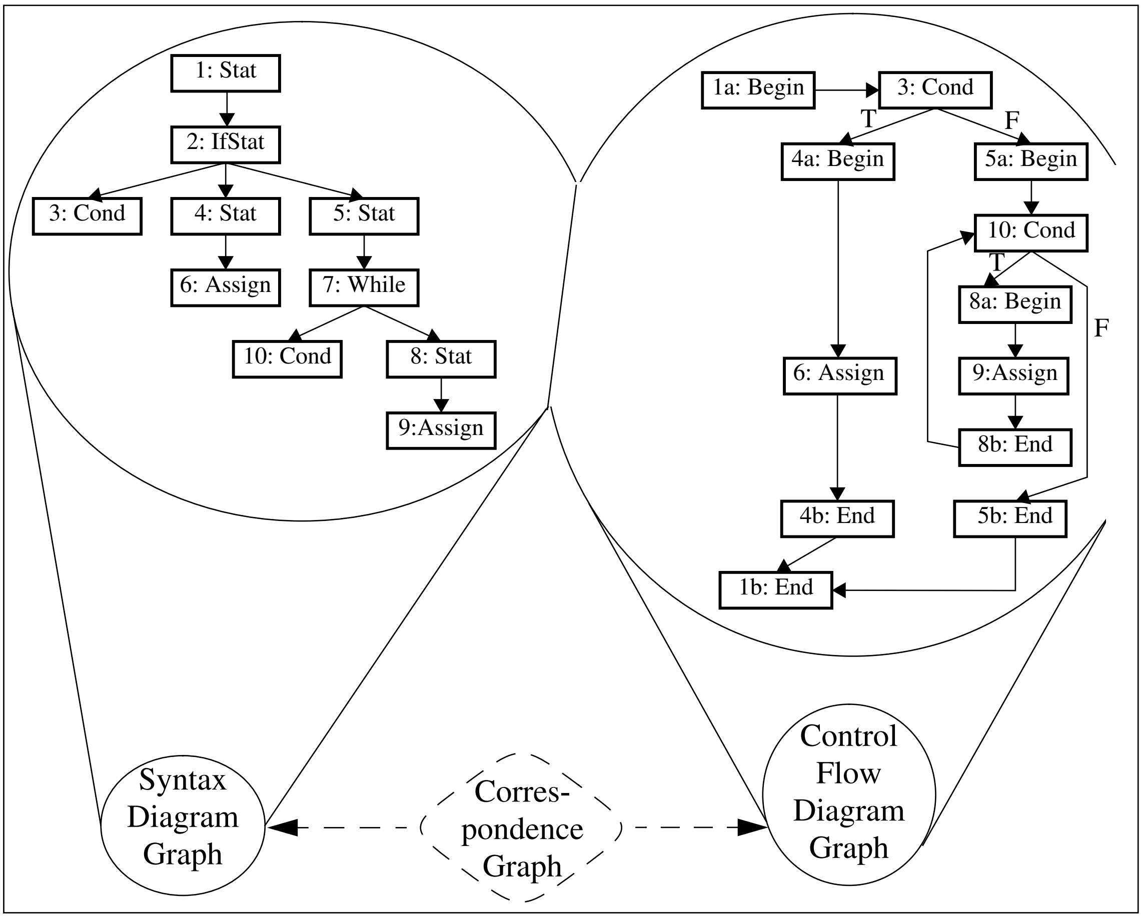

Fig. 1: Related Graphs: Syntax Tree and Control Flow Diagram.

Examples of related graph-like data structures are for instance the requirements and design documents of a piece of software or the syntax tree and control flow diagram for a program [2, 9, 11, 22, 23]. Figure 1 shows a cutout of a program's syntax tree and its corresponding control flow diagram that consists of an if-statement with a single assignment in its then-branch and a while-statement in its else-branch. Labels of edges are omitted in order to keep diagrams legible with the exception of those edges in the right-hand side control flow graph that have conditions as their sources: "T(rue)" and "F(alse)" are their labels and have the usual meaning. The example demonstrates our needs for *fine-grained intergraph relationships* with the following *characteristics*:

- Elements of one graph are related to distinct elements of another graph: in figure 1, a vertex with number x of the left-hand side graph is related to a vertex with the same number (if existent) in the right-hand side graph.
- Correspondences between vertices and edges of related graphs are at least 1-to-n: in figure 1, a vertex x of the left-hand side graph corresponds to vertices xa and xb (if existent) of the right-hand side graph.
- Related graphs contain referenced as well as private elements: in our running example, all vertices of the right-hand side graph are referenced, whereas all its edges are private, and the referenced part of the left-hand side graph are all its vertices with exception of 2 and 7.

Please note that it is often a matter of taste whether a distinct node of one graph is private or whether it is related to a whole subgraph of another graph (it is also reasonable to relate the currently private node 7 of the syntax diagram graph to nodes 8a, 8b, and 10 of the control flow diagram graph). The formalism, presented within the next sections allows for the definition of arbitrary *m-to-n intergraph relationships* and, thus, supports both possibilities.

In general, intergraph relationships themselves have annotations with information about ongoing translation or analysis processes. Sometimes, we have even relationships between intergraph relationships for context-sensitive dependencies like "this part of the source graph was translated before that part of the source graph" or "this intergraph relationship is only valid as long as that intergraph relationship is existent". These additional annotations and dependencies are of particular importance in the case of incrementally working translation or analysis processes, the description of which is outside the scope of this paper. Therefore, intergraph relationships are modeled as separate *correspondence graphs* with additional references to related source and target graph elements (cf. bottom part of figure 1).

The next section suggests a formalism for the specification of intergraph relationships and the accompanying analysis as well as translation processes in a *purely declarative way*, where the same specification serves as input for the development of a whole family of batch-oriented as well as incrementally working integration tools. Section 3 presents its accompanying theory and explains how to construct (batch-oriented) graph-to-graph translators. Section 4 discusses afterwards a number of necessary extensions for the presented approach and section 5 finally summarizes the main ideas and gives an outlook on future work.

2. Triple Graph Grammars and Running Example

Triple graph grammars are a refinement of the old idea of *pair graph grammars* already suggested by Pratt [18]. Pair graph grammars as well as (almost) all EBNF-oriented approaches [17, 22, 23] for tree-to-tree translations are restricted to context-free productions and one-to-one correspondences between objects in related data structures. Therefore, these formalisms are far too limited with respect to their expressiveness (see example of figure 1).

To compensate for these deficiencies triple graph grammars extend the original pair graph grammar approach to the case of *context-sensitive productions* with rather complex left- and right-hand sides, and offer separate correspondence rules and graphs for modeling *m-to-n-relationships* between related graphs. As already mentioned above, these correspondence graphs and rules allow us to record additional information about the transformation process itself, which are for instance needed to propagate incremental updates of one data structure as incremental updates into its related data structures (for further details see [11, 12]). The name *triple graph grammars* has been chosen in order to emphasize the important role of these additional correspondence rules and graphs.

Before going into details, we have to discuss one important "design decision", the chosen *representation of intergraph relationships* between corresponding graph elements and source or target graph elements. One idea, suggested by one of the paper's referees, is to embed all three graph components as substructures into a common "superstructure" and to use (higher order) relations for modeling correspondences between nodes (and edges!) of these substructures. This solution has the advantage that simple rewrite rules of high level replacement systems with well-known properties [4] are sufficient for the specification of graph-to-graph translation processes (in contrast to triple productions, introduced in definition 3.4). Nevertheless, it has a number of significant disadvantages:

- Relationships between a production's left- and right-hand side are modeled different than relationships between (at least conceptually) separate productions for separate source and target graphs, although playing a similar role within the following definitions.
- The data model of directed graphs must be abandoned in favor of a new data model which allows for the definition of relationships between relationships.
- Finally, this approach enforces the introduction of embedding superstructures even in the case where input and output of translation processes are viewed as separate graphs and where fine-grained correspondences are of no importance afterwards.

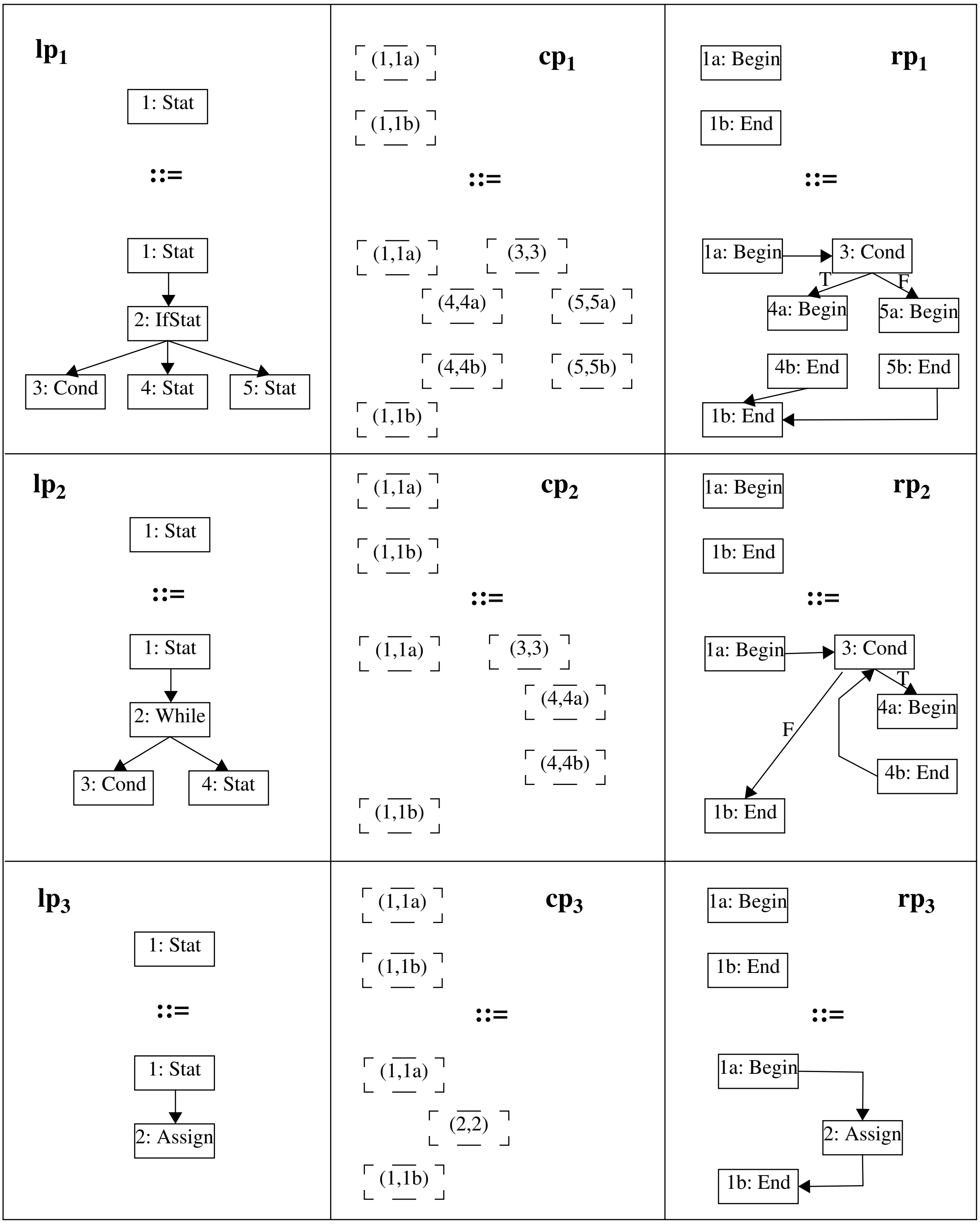

Fig. 2: Production Triples for Syntax Tree ↔ Control Flow Diagram Translations.

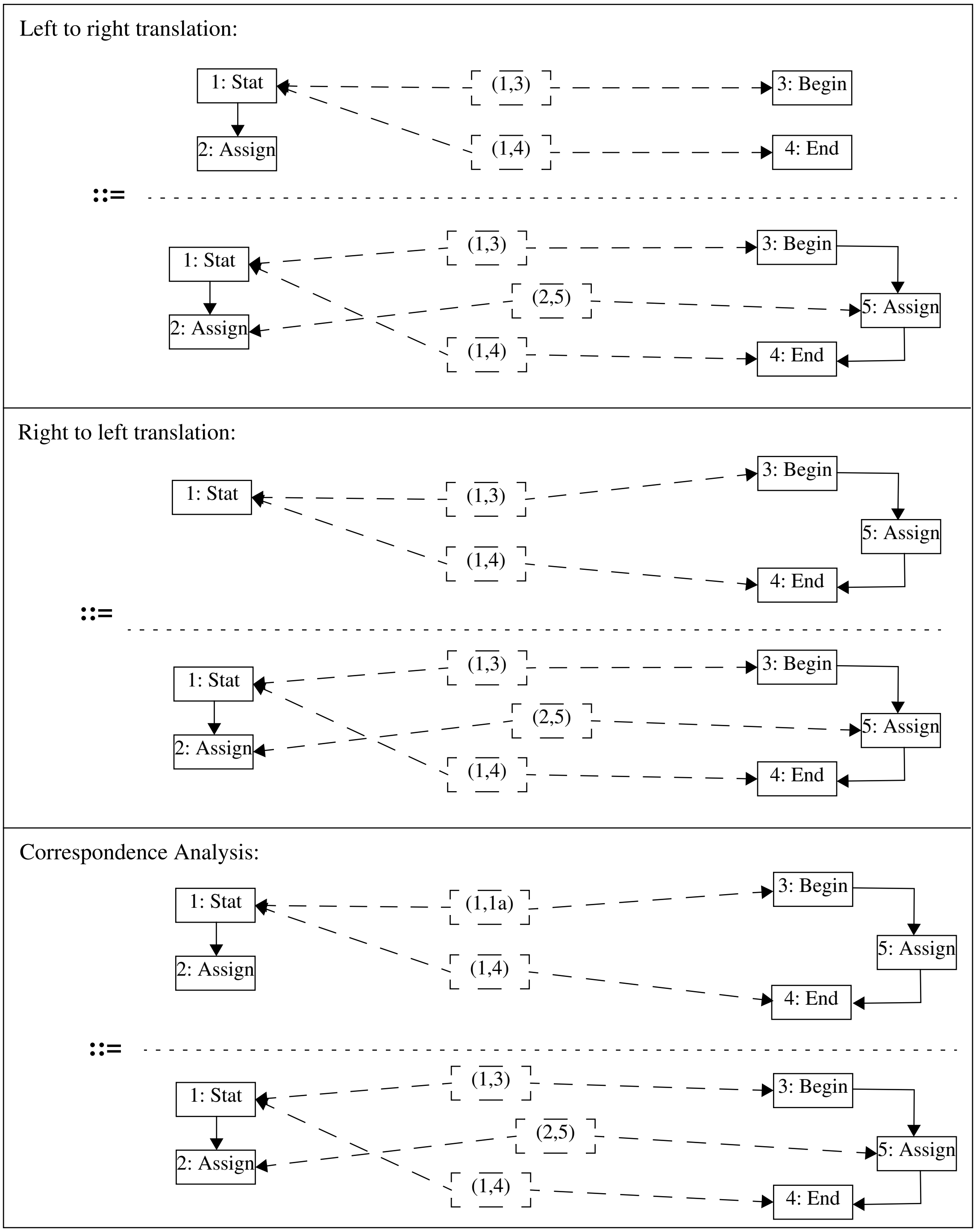

Fig. 3: Conventional Productions for Syntax Tree ↔ Control Flow Diagram.

Therefore, it seems to be more appropriate to use morphisms from correspondence graphs to source and target graphs for modeling intergraph relationships.

Returning to the example of section 1, we will now discuss its specification in figure 2. The first production triple defines the correspondences between an if-statement's syntax tree and its control flow (sub-)diagram, whereas following production triples deal with while-statements and assignments. Please note that a dashed vertex (x,y) of a correspondence production cp_i models a relation between a vertex x in the left-hand side syntax tree and a vertex y in the right-hand side control flow diagram of figure 1.

Due to lack of space, productions dealing with statement sequences are omitted and branches of if-statements as well as bodies of while-statements are confined to be single statements only. Last but not least pairs of new "Begin" and "End" vertices within right-hand side productions are intentionally left unconnected. In this way we are able to demonstrate that the language of all graphs generated by isolated right-hand side productions is a proper superset of the language of all control flow graphs components generated by our triple graph grammar. It contains entangled control flow diagrams, where single statements connect wrong pairs of "Begin" and "End" vertices. This reflects the fact that, in general, we cannot expect that all possible inputs of graph-to-graph translation processes are also legal inputs which have a defined result[1].

The presented specification may be used to develop batch-oriented as well as incrementally working tools of rather different functionality as for instance

- an *LR-translator*, which takes any left-hand side syntax tree as input and returns a corresponding right-hand side control flow diagram,
- an *RL-translator*, which analyzes a right-hand side control flow diagram and produces the corresponding syntax tree if possible, and
- a *correspondence analyzer*, which monitors the relationships between a given syntax tree and a given control flow diagram.

The advantage of triple graph grammars in comparison to previously used graph grammar approaches for the specification of intergraph relationships, as e.g. in [22], is clearly visible when we compare the triple production (lp_3, cp_3, rp_3) of figure 2 with its corresponding "conventional" productions of figure 3. In the latter case, we have a single "flat" and often unintelligible graph on left- and right-hand sides of productions (with correspondences highlighted as dashed nodes and edges and with left-hand sides above and right-hand sides below dotted lines). The first one specifies the corresponding LR-translation step, the second one the corresponding RL translation step, and the last one a correspondence analysis step which takes a syntax diagram and a control flow diagram as input and tries to establish correspondences between these graphs afterwards. That means that one triple production replaces three rather similar but nevertheless different conventional productions. The next section presents an algorithm which takes a set of triple productions as input and generates the corresponding sets of conventional productions.

3. Simple Triple Graph Grammars and LR-/RL-Translator

Triple graph grammars may be used to specify rather complex graph-to-graph translations as languages of graph triples. Elements (LG, CG, RG) belonging to these languages represent related graph structures LG and RG, respectively, which are linked to each other by means of an additional correspondence graph CG. The grammar for such a graph triple language consists of triples of productions (lp, cp, rp), where each production component is responsible for generating or extending the corresponding graph component.

1) Sometimes, specified translation processes are even *partial and nondeterministic* functions and additional user input is necessary to resolve conflicts between concurrently applicable production triples (see [12]).

In principle, any graph model and any graph grammar approach may be used as the fundament of triple graph grammars. To emphasize this, we will start with a very simple class of graphs and rather straightforward rewriting rules, such that we are able to explain the principles of the new formalism both within the framework of *the algorithmic and the algebraic graph grammar approach* [3, 14] without getting stuck into technical details. Afterwards, in section 4, we will sketch how to get rid of the limitations of the selected primitive graph model and the accompanying rewriting formalism.

As the reader may imagine, the development of tools which extend related graphs in parallel by simultaneously applying related productions to related vertices and edges is a straightforward task. But the development of tools which translate an existing left-hand (right-hand) side graph into a new right-hand (left-hand) side graph is a rather difficult task. In general, it requires the realization of an efficiently and preferably even incrementally working *graph parser* for context-sensitive productions which is able to recover a sequence of production applications yielding the given left-hand (right-hand) side graph. Provided with this sequence of productions we are then able to apply the related sequence of productions to a start graph and to produce thereby the required right-hand (left-hand) side graph.

In the presented version of triple graph grammars we circumvent this problem by regarding *monotonic productions* only. This means that any production's left-hand side must be part of its right-hand side, i.e. productions do not delete vertices and edges. In this case, a given graph directly contains all necessary information about its derivation history and the development of LR- or RL-translators is simplified considerably.

Requiring monotonicity is not as restrictive as it seems to be at a first glance, because triple graph grammars are not intended to model editing processes on related graphs (with insertions as well as deletions and modifications of graph elements, but are a generative description of graph languages and their relationships.

Following this line we start with the definition of simple graphs, graph morphisms, and monotonic productions:

Definition 3.1 Graphs, Graph Morphisms, and Graph Operators.

A quadruple G := (V, E, s, t) is a *graph* with *elem*(G) := $V \cup E$ and

(1) V being a finite[2] set of vertices,

(2) E being a finite set of edges, and

(3) s, t: $E \to V$ being two functions which assign source and target vertices to edges.

Let G := (V, E, s, t), G' := (V', E', s', t') be two graphs. A pair of functions h := (h_V, h_E) with h_V: $V \to V'$ and h_E: $E \to E'$ is a *graph morphism* from G to G',i.e. h: $G \to G'$, iff

(4) $\forall\, e \in E$: $h_V(s(e)) = s(h_E(e)) \,\wedge\, h_V(t(e)) = t(h_E(e))$.

Furthermore, the operators "$\subset$" for "proper subgraph", "$\subseteq$" for "subgraph", "$\cup$" for "union of graphs with gluing of identified nodes and edges (with same identifiers)", "$\cap$", and "\" , , "union", etc. are defined as usual for graphs, and with h: $G \to G'$ being a morphism, h(G) $\subseteq$ G' denotes that *subgraph* in G' which is the image of h. ■

2) Finiteness is required to guarantee termination of LR/RL-translations, see proposition 3.9.

Definition 3.2 Monotonic Productions and Graph Rewriting.

Any tuple of graphs p := (L, R) with $L \subseteq R$ is a *monotonic productio*n and p applied to a given graph G produces another graph $G' \supseteq G$, denoted by: G ~p~> G', with respect to redex[3] selecting morphisms $g: L \rightarrow G$ and $g': R \rightarrow G'$, iff:

(1) $g'|_L = g$, i.e. g and g' are identical mappings w.r.t. the left-hand side graph L.

(2) g' maps new vertices and edges of $R \setminus L$ onto unique new vertices and edges of $G' \setminus G$.

Using the categorical framework [3], conditions (1) and (2) may be replaced by requiring the existence of the pushout diagram a) of figure 3. ■

Based on this fundamental terminology we are now able to define graph triples as well as production triples and their application to graph triples:

Definition 3.3 Graph Triples.

Let LG, RG, and CG be three graphs, and $lr: CG \rightarrow LG$, $rr: CG \rightarrow RG$ are those morphisms which represent m-to-n relationships between the left-hand side graph LG and the right-hand side graph RG via the correspondence graph CG in the following way:

$$x \in LG \text{ is related to } y \in RG \;:\Leftrightarrow\; \exists\, z \in CG\!:\; x = lr(z) \;\wedge\; rr(z) = y \,.$$

The resulting *graph triple* is denoted as follows:

$$GT := (\, LG \leftarrow lr — CG — rr \rightarrow RG \,) \,.$$ ■

Definition 3.4 Production Triples and Graph Triple Rewriting.

Let lp := (LL, LR), rp := (RL, RR), and cp := (CL, CR) be monotonic productions. Furthermore, $lh: CR \rightarrow LR$ and $rh: CR \rightarrow RR$ are graph morphisms such that their restrictions $lh|_{CL}: CL \rightarrow LL$ and $rh|_{CL}: CL \rightarrow RL$ are morphisms, too, which relate the left- and right-hand sides of productions lp and rp via cp to each other. The resulting *production triple* is denoted as follows:

$$p := (\, lp \leftarrow lh — cp — rh \rightarrow rp \,) \,.$$

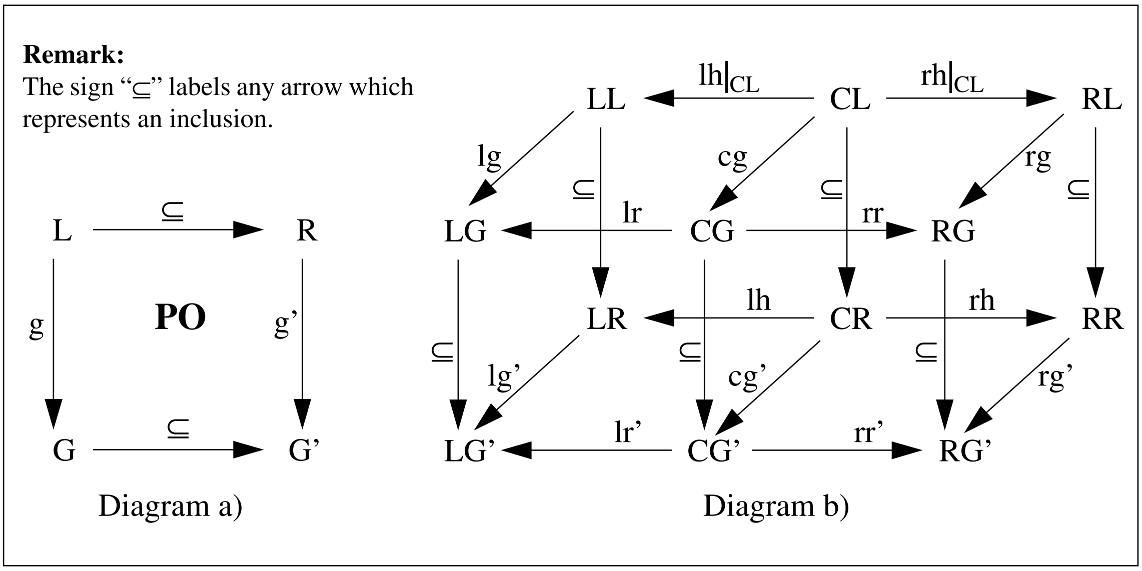

Fig. 4: Application of Simple Productions and Production Triples.

3) A redex is a subgraph within a host graph matching a given production's left-hand side.

And the application of such a production triple to a graph triple

$$GT := (LG \leftarrow lr — CG — rr \rightarrow RG)$$

produces another graph triple

$$GT' := (LG' \leftarrow lr' — CG' — rr' \rightarrow RG') ,$$

i.e.:

$$GT \sim p \sim> GT' ,$$

which is uniquely defined (up to isomorphism) by the existence of the “pair of cubes” diagram b) of figure 3. This diagram consists of commuting square-like subdiagrams only and contains a pushout subdiagram for each application of a production component to its corresponding graph component.

In the sequel, we often have to deal with production triple applications, where the redex or result for their left- or right-hand side production application is already known in the form of a morphism g. We denote these restrictions for rewriting GT into GT’ by

$$GT \sim p(g) \sim> GT'. \blacksquare$$

Proposition 3.5 Soundness of Graph Triple Rewriting.

With the abbreviations of definition 3.4 we have to prove the following property: The result of the application of a production triple is uniquely determined (up to isomorphism) by redex selecting morphisms (lg, cg, rg), i.e. the diagram of figure 4 with commuting subdiagrams has a unique completion to a “*pair of cubes*” as presented in figure 3.

Proof:

All morphisms with exception of those building the bottom sides of the pair of cubes are already fixed with the selection of a triple redex. Furthermore, (lg’, cg’, rg’) are determined by definition 3.2 and the right-hand side diagram of figure 5 proofs the existence and uniqueness of the missing graph morphism rr’: CG’ $\rightarrow$ RG’ such that

$$rr = rr'|_{CG} \quad \text{and} \quad rh \circ rg' = cg' \circ rr' .$$

This is a direct consequence of the pushout property for the square with corners CL, CG, CG’, and CR but can also be proved by explicitly constructing rr’:

$$rr'(x) = \underline{\text{if}}\ x \in CG\ \underline{\text{then}}\ rr(x)\ \underline{\text{else}}\ (cg'^{-1} \circ rh \circ rg')(x)\ \underline{\text{fi}} .$$

This a sound definition, because cg’ is injective for all elements in CG’ \ CG and for all elements $x \in CL$: $(cg \circ rr)(x) = (rh|_{CL} \circ rg)(x) = (rh \circ rg')(x)$

$$\Rightarrow \quad \forall x \in cg(CL) \subseteq CG: \quad rr(x) = (cg'^{-1} \circ rh \circ rg')(x) .$$

In just the same way we can prove that

$$lr'(x) = \underline{\text{if}}\ x \in CG\ \underline{\text{then}}\ lr(x)\ \underline{\text{else}}\ (cg'^{-1} \circ lh \circ lg')(x)\ \underline{\text{fi}} . \blacksquare$$

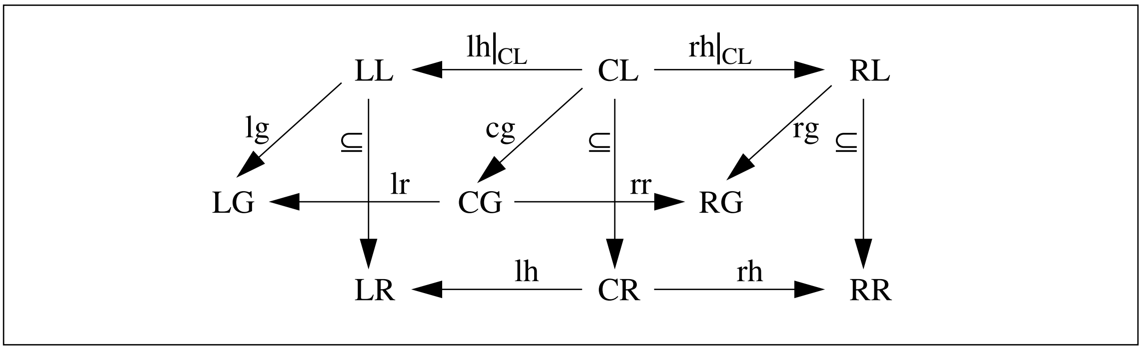

Fig. 5: Redex Selection for a Production Triple.

$$
\begin{tikzcd}
CL \arrow[rr, "\subseteq"] \arrow[dd, "cg"'] \arrow[ddrr, phantom, "\mathbf{PO}"] & & CR \arrow[dr, "lh"] \arrow[dd, "cg'"'] & \\
& & & LR \arrow[dd, "lg'"] \\
CG \arrow[rr, "\subseteq"] \arrow[dr, "lr"'] & & CG' \arrow[dr, dotted, "lr'"] & \\
& LG \arrow[rr, "\subseteq"] & & LG'
\end{tikzcd}
\qquad
\begin{tikzcd}
CL \arrow[rr, "\subseteq"] \arrow[dd, "cg"'] \arrow[ddrr, phantom, "\mathbf{PO}"] & & CR \arrow[dr, "rh"] \arrow[dd, "cg'"'] & \\
& & & RR \arrow[dd, "rg'"] \\
CG \arrow[rr, "\subseteq"] \arrow[dr, "rr"'] & & CG' \arrow[dr, dotted, "rr'"] & \\
& RG \arrow[rr, "\subseteq"] & & RG'
\end{tikzcd}
$$

Fig. 6: Existence and Uniqueness of lr' and rr'.

Having defined the application of production triples to graph triples we are now able to model processes which extend related graphs (and their interrelationships) *synchronously*. But how can we handle the case where a left-hand side graph is given and we have to construct the missing right-hand side graph including all intergraph relationships or vice versa?

For symmetry reasons, the solution is the same in both directions. Therefore, the construction of LR-translators will be discussed in detail and the solution for RL-translators may be obtained by simply exchanging the roles of “left”- and “right”-hand side components. Informally speaking we simply have to split a production triple p into a pair of production triples p_L and p_{LR}, where p_L is a *left-local production triple* which rewrites the left-hand side graph only and p_{LR} is a *left-to-right translating production triple* which keeps the new left-hand side graph unmodified but adjusts the correspondence and right-hand side graph such that for any morphism lg that restricts the application of left-hand side productions

$$p(lg) = p_L(lg) \circ p_{LR}(lg) \,,$$

i.e. the effect of applying first p_L and then p_{LR} to a given graph triple is the same as applying p itself if (and only if) we keep the left-hand side redex, i.e. the morphism lg, fixed.

Then, provided with a sequence of productions p_L^1 to p_L^n and a sequence of corresponding redex selecting morphisms lg^1 to lg^n that produce a certain left-hand side graph LG, we will be able to construct a corresponding right-hand side graph RG by executing p_{LR}^1 to p_{LR}^n .

Within the following propositions we will show that it is always possible to split a production triple into a left-local production and a left-to-right transformation. Furthermore, we will develop an always terminating algorithm which computes all graph triples containing a given left- or right-hand side graph.

Proposition 3.6 LR-Splitting of Production Triples.

A given production triple p := ((LL, LR) ← lh — (CL, CR) — rh → (RL, RR)) may be split into the following pair of equivalent production triples:

(1) p_L := ((LL, LR) ← ε — (∅, ∅) — ε → (∅, ∅)) is the left-local production for p, where ∅ is the empty graph and ε is an inclusion of the empty graph into any graph.

(2) p_{LR} := ((LR, LR) ← lh — (CL, CR) — rh → (RL, RR)) is the left-to-right translating production for p.

For these production triples and any graph triples GT := (LG ← lr — CG — rr → RG), GT' := (LG' ← lr' — CG' — rr' → RG'), and a morphism lg': LR → LG' the following proposition holds:

$$GT \sim p(lg') \sim> GT' \quad \Leftrightarrow \quad \exists\, HT: \; GT \sim p_L(lg') \sim> HT \;\wedge\; HT \sim p_{LR}(lg') \sim> GT' .$$

Proof:

The following equivalences prove that the vertical sides of the cubes of figure 3 b) and figure 6 imply each other if (and only if) all production applications use the same morphism lg' to select an image of graph LR in LG' (and thereby of LL in LG):

(1) LG ~(LL, LR)~> LG' ⇔ LG ~(LL, LR)~> LG' ∧ LG'~(LR, LR)~> LG' .

(2) CG ~(CL, CR)~> CG' ⇔ CG ~(CL, CL)~> CG ∧ CG ~(CL, CR)~> CG'
⇔ CG ~(∅, ∅)~> CG ∧ CG ~(CL, CR)~> CG' .

(3) RG ~(RL, RR)~> RG' ⇔ RG ~(RL, RL)~> RG ∧ RG ~(RL, RR)~> RG'
⇔ RG ~(∅, ∅)~> RG ∧ RG ~(RL, RR)~> RG' .

And proposition 3.5 guarantees existence and uniqueness of all horizontal arrows. Furthermore, diagram 3 b) is equivalent to

GT ~p(lg')~> GT'

and, if we merge the two rows of cubes in the upper part of diagram 6 to a single row of cubes, then the new upper part of diagram 6 is equivalent to

$GT \sim p_L(lg') \sim> HT$.

Finally, the lower part of diagram 6 is equivalent to

$HT \sim p_{LR}(lg') \sim> GT'$. ■

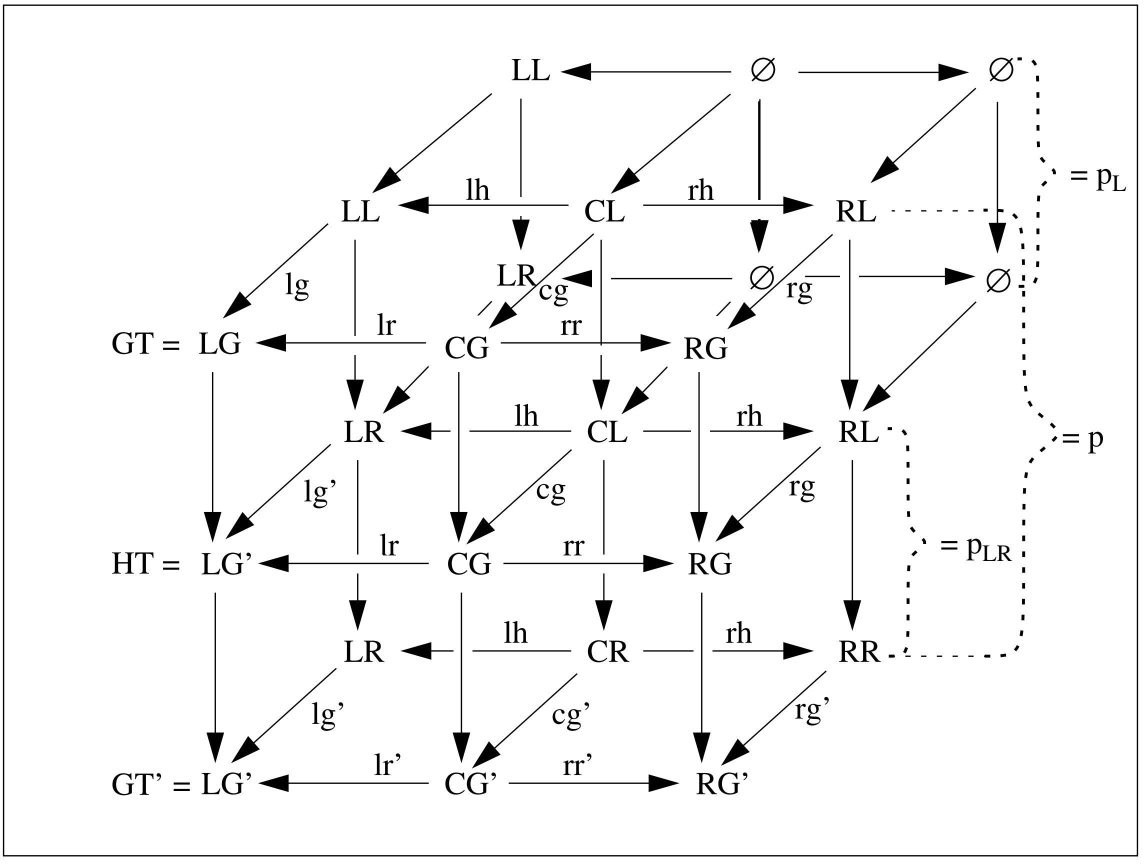

Fig. 7: Splitting of Production Triple Application.

Please note that we used the name “lr” in figure 6 to denote a morphism from CG to LG as well as its range extension to a morphism from CG to LG’ $\supseteq$ LG. Furthermore, all arrows without any label denote inclusions and the domain restrictions of lh and rh from CR to CL have been omitted in order to keep the diagram as legible as possible.

In a similar way the splitting of a production triple into a right-local production followed by a right-to-left translating production may be defined, but we have still to show that we can use these locally equivalent splittings for the definition of graph transformations which create first a left-hand side graph completely and add a corresponding right-hand side graph and the accompanying correspondence graph afterwards or vice versa, i.e. we have to prove:

Proposition 3.7 Permutation of Left-Local and Left-to-Right Productions.

Given n production triples p^1 through p^n and morphisms lg^1 to lg^n which determine the application results of left-hand side production components of p^1 through p^n we can prove that

$$p^1(lg^1) \circ \ldots \circ p^n(lg^n) \;=\; (\, p^1_L\,(lg^1) \circ \ldots \circ p^n_L\,(lg^n)\,) \circ (\, p^1_{LR}\,(lg^1) \circ \ldots \circ p^n_{LR}\,(lg^n)\,)\;.$$

Proof:

This follows directly from proposition 3.6 that ensures

$$p^1(lg^1) \circ \ldots \circ p^n(lg^n) \;=\; (\, p^1_L\,(lg^1) \circ p^1_{LR}\,(lg^1)\,) \circ \ldots \circ (\, p^n_L\,(lg^n) \circ p^n_{LR}\,(lg^n)\,)$$

and the fact that

(1) a production triple $p^k_L := (\,(LL, LR) \xleftarrow{\varepsilon} (\emptyset, \emptyset) \xrightarrow{\varepsilon} (\emptyset, \emptyset)\,)$ modifies left-hand side graph components only and has no requirements with respect to correspondence or right-hand side graphs,

(2) a simple production (LR, LR) may be applied to a graph LG’ without causing any modifications, whenever LG’ is the result of applying first a monotonic production (LL, LR) followed by an arbitrary number of different monotonic productions,

(3) and a production triple $p^k_{LR} := (\,(LR, LR) \xleftarrow{lh} (CL, CR) \xrightarrow{rh} (RL, RR)\,)$ keeps its left-hand side graph unmodified.

Therefore, we are allowed to exchange the application order of production triples freely as long as for any natural numbers $i \leq k$

(4) the application of p^i_L precedes the application of p^k_L for $i \neq k$,

(5) the application of p^i_{LR} precedes the application of p^k_{LR} for $i \neq k$,

(6) and the application of p^i_L precedes the application of p^k_{LR} . ■

In the same way we are able to prove that a sequence of production triples may replaced by an equivalent sequence of corresponding right-local and right-to-left translating productions, where all right-local productions build the first half of the sequence and all right-to-left translating productions the second half. Therefore, the problem of constructing LR- or RL-translations is solved in principle with the exception of the following two questions:

- How can we find a sequence of left-local (right-local) productions which creates a given left-hand (right-hand) side graph?
- Is the number of graph triples, which contain a given left-hand (right-hand) side graph, a finite set?

To answer these questions we have to restrict our interest to finite graphs and finite sets of production triples. Furthermore, these production triples are expected to fulfill an additional “safeness” criteria which allows us to prove termination of LR-translations later on:

Definition 3.8 LR-Safeness of Production Triples.

Using the abbreviations of definition 3.4 a production triple p is *LR-safe* iff

$new_{LR}(p) \neq \emptyset$, with $new_{LR}(p) := elem(LR) \setminus old_{LR}(p)$ and $old_{LR}(p) := elem(LL)$. ■

LR-safe production triples have strictly monotonic left-hand side productions (they have at least one new edge or node). Therefore, an upper boundary for the length of any production triple sequence is known that creates a given (finite) graph LG as the left-hand side of the resulting graph triple and the following algorithm may be used to compute LR-translations:

Proposition 3.9 Computing LR-Translations.

Using the abbreviations of definition 3.8 and proposition 3.7 and provided with a finite set $\mathcal{R} := \{ p^1, \ldots, p^n \}$ of LR-safe production triples and a graph LG, the finite set of all $\mathcal{R}$-generated graph triples with LG as their left-hand side graph component may be computed as follows (for simplicity reasons, empty graphs are used as axioms/start graphs):

```
CoveredElements := ∅;
HT := GT := ( LG ← ε — ∅ — ε → ∅ );
while elem(LG) \ CoveredElements ≠ ∅ do
    select p ∈ ℛ with left-hand side production (LL,LR);
    select morphism lg: LR → LG   with lg(new_LR(p)) ∩ CoveredElements ≠ ∅
                                  and  lg(old_LR(p)) ⊆ CoveredElements
                                  and  lg | new_LR(p) is injective
                                  and  HT ~p_LR(lg)~> HT';
    HT := HT';
    CoveredElements := CoveredElements ∪ lg(new(p));
end;
return GT' := HT as one possible LR-translation/completion of LG;
```

Proof:

The *correctness* of all results produced by the algorithm above is a direct consequence of proposition 3.7 if we take into account that

$$GT \sim(p^1_{LR} (lg^1) \circ \ldots \circ p^n_{LR} (lg^n))\sim> GT'$$

implies that

$$(\emptyset \leftarrow \varepsilon — \emptyset — \varepsilon \rightarrow \emptyset) \sim(p^1_L (lg^1) \circ \ldots \circ p^n_L (lg^n))\sim> GT$$

because

(1) $lg^1(new(p^1)) \cup \ldots \cup lg^n(new(p^n)) = elem(LG)$,

(2) $lg^i(new(p^i)) \cap lg^j(new(p^j)) = \emptyset$, for $i \neq j$, and

(3) $lg^i(x) \neq lg^i(y)$, for $x, y \in new(p^i)$.

Furthermore, the algorithm's *termination* is guaranteed by finiteness of LG (see definition 3.1), finiteness of $\mathcal{R}$, and continuous growth of the set CoveredElements (definition 3.8 requires that $new(p) \neq \emptyset$). But the algorithm's *completeness* is a serious problem due to the nondeterministic selection of productions and morphisms in lines 4 and 5. These selections are locally correct but may lead the overall translation process into dead-ends. Therefore, we have to transform the suggested nondeterministically working algorithm into either a breadth-first search algorithm, which constructs all possible derivation sequences in parallel, or into a depth-first search algorithm, which constructs one derivation sequence after the other and uses backtracking to escape out of dead-ends. ■

Before turning our interest to useful extensions of the presented triple graph grammar approach within the next section, we have to add the following remarks:

- Proposition 3.9 explains only how to realize batch-oriented LR- and RL-translator*s*. But in a similar fashion *consistency checking tools* may be constructed that take a pair of left- and right-hand side graph components as input and try to produce the missing correspondence graph as output.
- Additional efforts are necessary to realize *incrementally working tools* for LR-/RL-translations or consistency checks. In this case, the correspondence graph has to contain additional information about executed translation steps in order to be able to recognize invalid subgraphs within their outputs efficiently, withdraw these invalid parts, and to reapply distinct LR- or RL-translating productions to reestablish consistency between related graphs (for further details cf. [11, 12]).
- Finally all implemented versions of LR-translators avoid the above mentioned completeness problem by requiring *additional confluence/uniqueness properties* for a set of production triples $\mathcal{R}$ (cf. [12]). But forthcoming realizations of LR- or RL-translators will take advantage of rather efficiently working backtracking mechanisms which are currently used within the graph-rewriting language PROGRES to "undo" effects of failing graph transformation sequences (cf. [24]).

4. Extended Triple Graph Grammars

When studying applications like the specification of tools that translate software requirement documents into software design documents or software design documents into program templates, we soon recognize that the presented triple graph grammar formalism of section 3 is still too restricted to be of any *practical relevance*. Even our running example exceeds the limits of the suggested primitive graph model by having vertex and edge labels as well as vertex attributes in an extended version. There is a significant gap between definitions 3.1 and 3.2 of this paper on one side and the graph model and rewriting approach supported by the application-oriented graph grammar language PROGRES on the other side [19].

It is the purpose of this section to *improve expressivenes*s of the graph model and the rewriting approach presented here as far as possible without destroying the proof of proposition 3.9. We will achieve this goal in the following steps:

(1) First of all we have to introduce vertex and edge labels.
(2) Then we have to deal with attributes for vertices (and edges).
(3) And finally, we have to introduce additional means for restricting the applicability of productions.

To solve task (1) we have to introduce sets $\mathcal{L} := \mathcal{L}_V \cup \mathcal{L}_E$ of *vertex* and *edge labels* for each regarded class of graphs, labeling functions l_V: $V \to \mathcal{L}_V$, l_E: $E \to \mathcal{L}_E$ for any graph

$$G := (V, E, s, t) \text{ over the label set } \mathcal{L},$$

and additional *compatibility relations* $\sim_{\mathcal{L},\mathcal{L}'} \subseteq \mathcal{L} \times \mathcal{L}'$ between label sets $\mathcal{L}$ and $\mathcal{L}'$. These relations restrict the definition of a morphism $h := (h_V, h_E)$ between the graph G above and another graph

$$G' := (V', E', s', t') \text{ over the label set } \mathcal{L}'$$

in the following way:

$$\forall\, v \in V\colon\ l_V(v) \sim_{\mathcal{L},\mathcal{L}'} l_{V'}(h_v(v)) \quad \wedge \quad \forall\, e \in E\colon\ l_E(e) \sim_{\mathcal{L},\mathcal{L}'} l_{E'}(h_E(e))\,.$$

Naturally, compatibility relations on $\mathcal{L} \times \mathcal{L}$ are expected to be identities, i.e. morphisms between graphs of the same class have to preserve vertex and edge labels, whereas additional application knowledge is necessary to define these relations between different label sets $\mathcal{L}$ and $\mathcal{L}'$ appropriately. Having a closer look onto the definitions and propositions of section 3, the introduction of labels raises one minor problem: Provided with certain sets of labels for left- and right-hand side graphs $\mathcal{LL}$ and $\mathcal{LR}$, how can we construct a label set $\mathcal{LC}$ for auxiliary correspondence graphs, and what about the definition of compatibility relations between left- or right-hand side graph labels and these auxiliary correspondence graph labels? A straightforward solution for this problem is to define $\mathcal{LC} \subseteq \mathcal{LL} \times \mathcal{LR}$ and to require

$$x \sim_{\mathcal{LL},\mathcal{LC}} (x, y) \sim_{\mathcal{LC},\mathcal{LR}} y\,.$$

For our running example $\mathcal{LC}$:= { (Stat,Begin), (Stat,End), (Assign,Assign), (Cond,Cond) }. In this way, our approach subsumes the definition of so-called EBNF correspondences in [17, 22] and is even closely related to a so-called "meta-modeling" approach in [9], where correspondences between object types of different diagram languages play the key role for the systematic development of diagram-to-diagram integration tools.

To solve task (2) mentioned above, we simply have to add further functions from vertices and edges onto given *attribute domains*. In this case the adaptation of definition 3.1 is straightforward by requiring that graph morphisms preserve attribute values.

But we have to solve another problem, the introduction of *attribute value parameters* and attribute value assignments for productions in order to be able to create graphs with an infinite number of possible attribute values without needing an infinite number of productions. The necessary extension of definition 3.2 may be performed in the same way as in [20] and splitting of a parametrized production p into p_L and p_{LR} (or p_R and p_{RL}) works as follows: The production p_L (p_R) has the same parameter list as p and is required to assign any parameter of this list to at least one attribute in the resulting left-hand (right-hand) side graph without overwriting old attribute values. And the remaining production p_{LR} (or p_{RL}) transfers attribute values from its left-hand (right-hand) side graph to its right-hand (left-hand) side graph, where necessary.

The remaining problem (3) of our list is a hard one. Currently we are trying to characterize a sufficiently large set of *application conditions* for permissible embeddings of a production's left-hand side into a host graph, which remain valid after the application of certain sequences of productions. In this way we intend to preserve the freedom to permute production applications further on (see proposition 3.7). All application conditions, for instance, which test the existence of additional vertices or edges in a host graph remain valid after the application of an arbitrary sequence of monotonic productions. But case studies in [2, 11, 12, 23] already indicate that the henceforth admissible class of application conditions is too small to express all necessary ordering constraints for concurrently applicable productions.

Therefore, we are currently introducing additional *production priorities* that prevent the application of productions in the presence of applicable productions with higher priorities. This solution is obviously less flexible than the usage of programs for controlling graph rewriting processes but has the clear advantage to preserve the envisaged declarative nature of triple graph grammars.

5. Conclusion

To summarize, *triple graph grammars* are a new formalism for the specification of complex interdependencies between separate and, in general, quite different graph-like data structures. Comparing them with previously suggested formalisms for the specification of data transformations they have the following advantages:

- The underlying data model are graphs and not trees as e.g. in [1, 7, 13, 17].
- The same specification may be used as the description of a unidirectional as well as a bidirectional transformation process in contrast to [1, 7, 13].
- Furthermore, correspondences are modeled explicitly and are not restricted to the case of 1-to-1 relationships between rather similar data structures as e.g. in [17, 22, 23].

Although being *purely declarative*, a triple graph grammar may be used as input for the production of a whole family of *rather efficiently working tools* of varying functionality as

- *batch-oriented translators* that take one data structure as input and return another new related data structure as output,
- *incrementally working translators* that propagate changes of one data structure into its related data structures and work in both directions,
- or even *consistency observing analyzer*s that infer or maintain correspondences between related parts of different data structures only.

But note that due to lack of space we had to concentrate on batch-oriented transformation tools in section 3 and to omit the construction of incrementally working transformation tools as wells as consistency observing analyzers. For more details on these subjects see [12]. Furthermore, we have to admit that all extensions proposed in section 4 are still on a more or less informal level and that all necessary adjustments of definitions and propositions of section 3 have still to be worked out in detail. Finally, we would like to get rid of the somewhat artificial restriction to monotonic productions by combining the presented approach with a theory of graph parsers for context-sensitive productions, although we are able to specify all kinds graph-to-graph translations, studied up to now, by means of monotonic triple graph grammars only.

References

[1] Aho A.V., Ganapathi M., Tijang S.W.K.: *Code Generation Using Tree Matching and Dynamic Programming*, in: acm Transactions on Programming Languages and Systems, vol. 11, no. 4, New York: acm Press (1989), 491-516

[2] Börstler J., Janning Th.: *Traceability Between Requirements Engineeering and Design: A Transformational Approach*, in: Proc. COMPSAC 92, Los Alamitos: IEEE Computer Society Press (1992), 362-368

[3] Ehrig H.: *Introduction to the Algebraic Theory of Graph Grammars (a Survey)*, in: Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology, LNCS 73, Berlin: Springer Verlag (1979), 1-69

[4] Ehrig H., Habel A., Kreowski H.J., Parisi-Presicce F.: *From Graph Grammars to High Level Replacement Systems*, in: Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532, Berlin: Springer Verlag (1991), 269-291

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

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

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

[8] Grosch J.: *Puma - A Generator for the Transfomation of Attributed Trees*, Compiler Generation Report no. 26, GMD Forschungsstelle an der Universität Karlsruhe (1991), Germany

[9] Himsolt M.: *GraphED: An Interactive Graph Editor*, in: Proc. STACS 89, LNCS 349, Berlin: Springer Verlag (1988), 532-533

[10] Janning Th.: *Integration von Sprachen und Werkzeugen zum Requirements Engineering und Programmieren im Großen*, Braunschweig: Deutscher Universitätsverlag (1992)

[11] Löwe M., Beyer M.: *AGG - An Implementation of Algebraic Graph Rewriting*, in: Proc. 5th Int. Conf. on Rewriting Techniques and Applications, LNCS 690, Berlin: Springer Verlag (1993), 451-456

[12] Lefering M.: *Tools to Support Life Cycle Integration*, in: Proc. 6th Software Engineering Environments Conference 1993 (SEE 93), Los Alamitos: IEEE Computer Society Press (1993), 2-16

[13] Lefering M.: *Development of Incremental Integration Tools Using Formal Specifications*, Technical Report AIB-94-2, RWTH Aachen, Fachgruppe Informatik, Germany (1994)

[14] Lipps P., Möncke U., Wilhelm R.: *OPTRAN - A Language/System for the Specification of Program Transformations, System Overview and Experiences*, LNCS 371, Berlin: Springer Verlag (1989), 52-65

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

[16] Nagl M.: *Graph Technology Applied to a Software Project*, in: The Book of L, Berlin: Springer Verlag (1986), 303-322

[17] Nagl M., Schürr A.: *A Specification Environment for Graph Grammars*, in: Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532, Berlin: Springer Verlag (1991), 599-609

[18] Normark K.: *Transformations and Abstract Presentations in a Language Development Environment*, Ph.D. Thesis, University of Aarhus, Denmark (1987)

[19] Pratt T.W.: *Pair Grammars, Graph Languages and String-to-Graph Translations*, in: Journal of Computer and System Sciences, vol 5, San Diego: Academic Press (1971), 560-595

[20] Schürr A.: *PROGRES: A VHL-Language Based on Graph Grammars*, in: Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532, Berlin: Springer Verlag (1991), 641-659

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

[22] Schürr A.: *Logic Based Structure Rewriting Systems*, in: Proc. Dagstuhl-Seminar 9301 on Graph Transformations in Computer Science, LNCS 776, Berlin: Springer-Verlag (1994), 341-357

[23] Westfechtel B.: *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*, IFB 280, Berlin: Springer Verlag (1991)

[24] Westfechtel B.: *A Graph-Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents*, in: Proc. Fifth International Workshop on Computer-Aided Software Engineering, Los Alamitos: IEEE Computer Society Press (1992), 2-13

[25] Zündorf A.: *Implementation of the Imperative/Rule Based Language PROGRES*, Technical Report AIB 92-38, RWTH Aachen, Germany (1992)