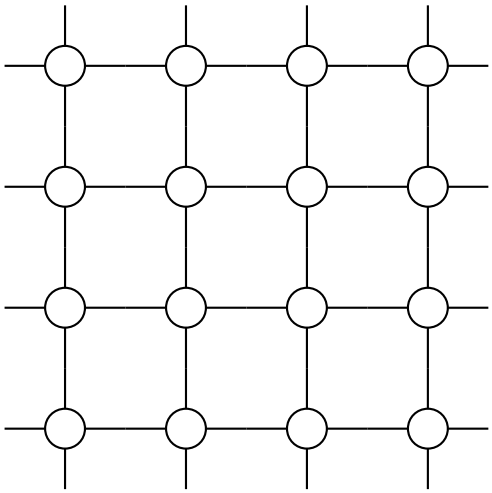
Annegret Habel
Mohamed Mosbah (Eds.)

Graph Computation Models

Second International Workshop, GCM 2008
Leicester, United Kingdom, September 2008
Proceedings



Preface

GCM 2008 is the second workshop of a series that serves as a forum for researchers that are interested in graph computation models. The scope of the workshop concerns graph computation models on graphical structures of various kinds (like graphs, diagrams, visual sentences and others). A variety of computation models have been developed using graphs and graph transformations. These models include features for programming languages and systems, paradigms for software development, concurrent calculi, local computations and distributed algorithms, biological or chemical computations. Graph transformations can be an intermediate representation of a computation. In addition to being visual and intuitive, this representation also allows the use of mathematical methods for analysis and manipulation.

The aim of the workshop is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation techniques, and their applications. A particular emphasis is made for models and tools describing general solutions.

The workshop includes tutorials and invited papers, contributed papers, and system demonstrations. The tutorials and invited papers introduce different types of graph transformations and their use to study computation models. The contributed papers consider specific topics of graph transformation models and their applications. The topics of the papers range from sequential graph transformation, extended graph rewrite rules, efficient pattern matching of the left-hand side of a rule to a host graph, and the termination of a controlled graph transformation system to parallel graph transformations in distributed adaptive design. The system demonstrations based on graph computation models range from alpha-versions to fully developed products that are used in education, research or being prepared for commercialisation. Accordingly, the proceedings of GCM 2008 consists of three parts. The first part comprises the extended abstracts of the tutorial and invited talks, the second part presents the contributed papers, and the third part contains extended abstracts of existing systems.

We would like to thank the members of the program commitee and the secondary reviewers for their enormous help in the selection process. Moreover, we would like to express our gratitude to the local organizers Reiko Heckel (Chair) and Dénes Bisztray (Workshop chair) who did a great job.

July 2008

Annegret Habel and Mohamed Mosbah
Program Chairs
GCM 2008

Organization

GCM 2008 is organized as a satellite workshop of the 4th International Conference on Graph Transformation (ICGT 2008), Leicester (United Kingdom), September 7 - 13, 2008 and takes place on September 8, 2008.

Program commitee

Frank Drewes	Umea (Sweden)
Rachid Echahed	IMAG, Grenoble (France)
Emmanuel Godard	Marseille (France)
Stefan Gruner	Pretoria (South Africa)
Annegret Habel (Co-chair)	Oldenburg (Germany)
Dirk Janssens	Antwerp (Belgium)
Hans-Jörg Kreowski	Bremen (Germany)
Mohamed Mosbah (Co-chair)	Bordeaux 1(France)
Detlef Plump	York (United Kingdom)

Table of Contents

Tutorial and Invited Talks

Full Papers

Position Papers

Systems Demonstrations

Modelling computational and logistic processes by autonomous units

Hans-Jörg Kreowski

University of Bremen

Abstract. The notion of a community of autonomous units is a graph-transformational device for the modelling of computational and - in particular - logisitic processes that co-exist, run and interact in the same environment. The units may communicate and cooperate with each other, their process runs may be sequential, parallel, or concurrent. In the talk, the framework will be introduced and some computational aspects stressed.

From Actors to Aspects: Programming with Graph Transformations

Dirk Jannsens

University of Antwerp

Abstract. The obvious advantage of computational models that directly manipulate discrete structures or graphs is that they allow one to work on a convenient level of abstraction, close to intuition about networks, object diagrams, or system states in general, and avoiding cumbersome coding into strings or formulas. In Graph Transformation systems, one takes the view that the desired manipulations can be obtained though local changes, embodied by rules: not only is the application of rules controlled by pattern matching, but it is also assumed that the changes are restricted to the pattern; i.e. the part of the structure that is not involved in a rule application remains unchanged.
The tutorial gives an overview of opportunities and challenges related to the use of graph transformation as a model of computation. As a starting point, work concerning Actor languages is used, which is based on very simple mathematics and should hence be accessible to a public not acquainted with the existing theory of graph transformation. Then issues concerning concurrency and modularity are discussed, as well as the challenges caused by the need to introduce more control and genericity than present in the basic mechanism. Finally we discuss how proposed solutions to these challenges are used in recent work about refactoring and model transformation, and we outline some further potential applications to aspect-oriented or delegation-based languages, as well as to topics outside of traditional computing science, such as self-assembly or natural computing.

An Introduction to GP

Detlef Plump

The University of York

Abstract. GP is a rule-based, nondeterministic programming language for solving graph problems at a high level of abstraction, freeing programmers from dealing with low-level data structures. GP's design aims at syntactic and semantic simplicity, to facilitate formal reasoning about programs. The language core consists of just four constructs: single-step application of a set of graph-transformation rules, sequential composition, branching and looping. This talk introduces GP by a number of example programs, presents a formal semantics in the style of Plotkin's structural operational semantics, and briefly describes the current implementation. Particular attention will be given to GP's powerful branching and looping constructs which allow to hide destructive tests and to iterate arbitrary subprograms.

Local Computations in Graphs: Impact of Synchronization on Distributed Computability

Jeremie Chalopin

University of Marseille

Abstract. In this talk, I will present different distributed models that can be expressed by graph relabelling systems. In these models, a computation step can be described by the application of a local relabelling rule that enables the modification of the states of neighboring vertices. These models represent different level of synchronization between adjacent processes.
We are interested in the computational power of these different models. In order to highlight the differences between these models, we study two classical problems in distributed computing: naming and election. The study of these problems enables to give a hierarchy between the different models and it enables to understand what kind of results are general enough to be expressed in each model.

Graph Rewrite Rules with Structural Recursion

Berthold Hoffmann[1], Edgar Jakumeit[2], and Rubino Geiß[2 3]

[1] Universität Bremen and DFKI-Lab Bremen, Germany
[2] Universität Karlsruhe (TH), Germany
[3] LPA GmbH Frankfurt/Main, Germany

Abstract. Graph rewrite rules, programmed by sequencing and iteration, suffice to define the computable functions on graphs—in theory. In practice however, the control program may become hard to formulate, hard to understand, and even harder to verify. Therefore, we have extended graph rewrite rules by variables that are instantiated by a kind of hyperedge replacement, before the so instantiated rules are applied to a graph. This way, rules can be defined recursively over the structure of the graphs where they apply, in a fully declarative way. Generic rules with variables and recursive rule instantiation have been implemented in the graph rewrite tool GrGen.

1 Introduction

Graph rewriting is a basis for rule-based ("declarative") programming with graphs, in the same way as term rewriting is a basis of functional programming—another rule-based paradigm. The following example from biology illustrates essential concepts of functional programming. We take this as a starting point for discussing concepts that would be useful for rule-based programming with graphs, and shall come back to it later.

Example 1 (Transcribing DNA to RNA). The genetic information of DNA is coded in four nucleic bases guanine (G), cytosine (C), adenine (A), and thymine (T), where uracil (U) replaces thymine in RNA. These bases form pairs G–C and A–T/U. A transcription of DNA to RNA starts after a sequence "TATAAA" on the DNA strand, and builds an RNA strand with complementary bases, until the termination sequence "CCCACT...AGTGGGAAAAAA" is found (where "..." stands for six arbitrary bases).

The following Haskell function defines transcription on strings representing the base sequences.

```
transcription ds
  | length ds < 30 = []
  | isTATAAA ds = d2rna ((drop 6) ds)
  | otherwise = transcription (tail ds) where
                d2rna ds | length ds < 24 = error "unterminated gene"
                         | isCCCACTuvwxyzAGTGGGAAAAAA ds = []
```

```
                | otherwise = d2r (head ds) : d2rna ( tail  ds)
d2r 'A' = 'U'; d2r 'C' = 'G'; d2r 'T' = 'A'; d2r 'G' = 'C';
```

(The omitted functions "isTATAAA" and "isCCC...AAA" test whether their arguments begin with the corresponding bases, and (drop *i*) removes i leading elements from a list.)

A rule-based functional language offers the following concepts:

1. A function may be defined with several rules that use *pattern matching* and *application conditions* for case distinction.
2. Patterns may contain *variables* like *ds*, which are placeholders for values with a specific, possibly recursive *structure*—character lists in this case.
3. Functions are defined by *recursion over the structure* of values. In our example, d2rna calls d2r on the head, and itself recursively on the tail of its argument.

Graph rewrite rules do certainly provide pattern matching, and may also support application conditions. However, in contrast to term rewriting, graph rewrite rules do not support variables that are placeholders for graphs of a specific structure. In most cases, structural recursion is not supported either. Instead, several graph rewrite tools feature constructs for choosing a rule from a set, sequential composition, and iteration. This suffices to define all computable functions on graphs [13]. However, are these concepts adequate from a programmer's point of view? They do suggest a style of programming that is imperative rather than declarative. More importantly, they certainly allow to control *which* rule shall be applied next, but provide only little help to control the places *where* it shall be applied.

Considering these deficiencies, we have extended the graph-rewrite tool GrGen [2] by generic rules with variables, where structure rules define the graphs that may be substituted for variables. Several structure rules may define alternative substitutions of a variable, and the substitutions may contain variables again, also recursively. So, a generic rule is instantiated recursively over a graph structure before it is applied. Variables may be placeholders for *sub-patterns* of a generic rule, like *ds* is a placeholder for string values. However, they may as well denote a *sub-rule*, like d2rna and d2r denote auxiliary functions in the example above. This concept shall improve the support for a declarative style of programming with graphs.

The paper is structured as follows. In the next section, the concepts of single-pushout (SPO) rewriting with negative application conditions are recalled. In Section 3, controlled graph rewriting is discussed. The limitations of these control programs have motivated our extension of rules by variables that are substituted according to recursive structure rules, which is described in Section 4. Finally, some related and future work is outlined in Section 5.

2 Graph Rewriting

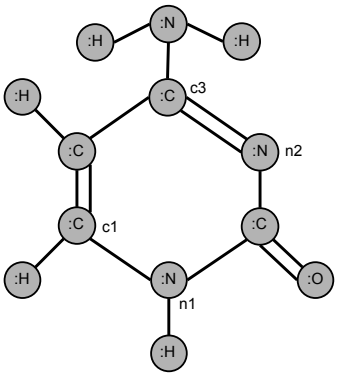
In this section, we review the major notions of graphs and rules implemented in the graph rewrite generator GrGen which is fully documented in [2] and [8].

Graphs. The graphs used in GrGen are directed and allow loops and multiple edges from one node to another one. Their nodes and edges are labeled (typed). Undirected edges are supported too, but for conciseness we want to view them as a shorthand notation for pairs of undirected counter-parallel edges in this paper. A fixed pair $T = (\dot{T}, \bar{T})$ of disjoint finite sets provides *types* for nodes and edges. A (typed) graph $G = (\dot{G}, \bar{G}, src_G, tgt_G, \dot{\tau}_G, \bar{\tau}_G)$ consists of disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *edges*, with mappings $src_G, tgt_G \colon \bar{G} \to \dot{G}$ that associate a *source* and a *target* node to its edges, and *type mappings* $\dot{\tau}_G \colon \dot{G} \to \dot{T}$ and $\bar{\tau}_G \colon \bar{G} \to \bar{T}$. We often write "$x \in G$" instead of "$x \in \dot{G}$ or $x \in \bar{G}$" and call x an *item* of G.

Let G and H be graphs. A pair $m = (\dot{m}, \bar{m})$ of functions $\dot{m} \colon \dot{G} \to \dot{H}$ and $\bar{m} \colon \bar{G} \to \bar{H}$ is a *graph morphism* (or just *morphism*, for short) if it preserves sources, targets, and types, i.e., if $src_H \circ \bar{m} = \dot{m} \circ src_G$, $tgt_H \circ \bar{m} = \dot{m} \circ tgt_G$, $\dot{\tau}_G = \dot{\tau}_H \circ \dot{m}$, and $\bar{\tau}_G = \bar{\tau}_H \circ \bar{m}$. Then m is denoted as $m \colon G \to H$, and called *injective* (*surjective* resp.) if its component mappings have this property. If m is injective and surjective, G and H are *isomorphic*, denoted as $G \cong H$.

We say that a graph G is a *subgraph* of a graph H, and write $G \subseteq H$, if the nodes and edges of G are subsets of those of H, and the mappings of G are restrictions of the respective mappings of H to $\bar{G}$ and $\dot{G}$.

Let G be a graph with a subgraph $D \subseteq G$. A morphism $m \colon D \to H$ is called a *partial morphism* from G to H, written $m \colon G \dashrightarrow H$, and D is called the *domain* of m, denoted by $Dom(m)$. The partial morphism m is *total* if $Dom(m) = G$.




```
node class C; // Program model
node class H;
node class N;
node class O;
```

```
test cytosine {
  n1:N -- c1:C -- c2:C;
  c2 -- c3:C -- n2:N -- c4:C -- n1;
  c2 -- c1; n2 -- c3;
  n1 -- :H; c1 -- :H; c2 -- :H;
  c3 -- n3:N -- :H; n3 -- :H;
  c4 -- o:O -- c4;
}
```

Fig. 1. The molecular structure of cytosine

Example 2 (Graphs). In Figure 1, the molecular structure of the nucleic base cytosine is specified in GrGen (on the right-hand side), and as a diagram (on the left-hand side). The GrGen program model on top declares four nodes types representing atoms, which are extensions of the predefined node type `Node`. Undirected edges of the predefined type `UEdge` represent chemical bonds. In the graph specification[4] below, items are introduced with $x : t$, where x is an optional item identifier, and t its type; items may be reused with their name. An undirected edge e with source x and target y is introduced by "x `-` e `-` y", and "`--`" introduces an anonymous edge of type `UEdge`.

In diagrams of graphs, nodes are depicted as circles, and edges are drawn as arrows from their source to their target nodes, undirected edges without tips. The type will be inscribed to the circle of a node, and ascribed to the arrow of an edge. Sometimes, node identifiers are ascribed to their circles.

Rewriting. GrGen is based on rewrite rules according to the single-pushout approach (SPO for short, see [17] for details) that may have negative application conditions as defined in [11].

A *graph rewrite rule* (*rule*, for short) is an injective partial morphism $r\colon P \dashrightarrow R$. A *conditional rule* is a pair (C, r) with r as above, and a set $C = \{c_1, \ldots, c_k\}$ of injective morphisms $c_i\colon P \to \tilde{P}_i$ (with $1 \leqslant i \leqslant k$). The graphs $\tilde{P}_i$ are *negative patterns*, P is the *pattern*, and R is the *replacement* of (C, r).

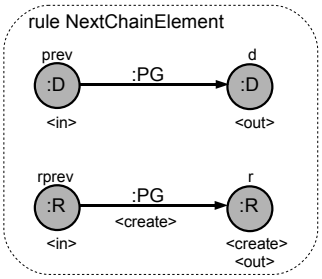
An injective total morphism $m\colon P \to G$ is a *match* of a conditional rule (C, r) as above if for all $c\colon P \to \tilde{P}$ in C there is no total injective morphism $\tilde{m}\colon \tilde{P} \to G$ so that $\tilde{m} \circ c = m$. A *rewrite step* of G using (C, r) via a match m yields a graph G' that is defined as a pushout, and can be constructed from the disjoint union of G and R by *(i) identifying*, for all $x \in Dom(r)$, the items $r(x)$ and $m(x)$, and *(ii) deleting*, for every $x \in P \setminus Dom(r)$, the item $m(x)$, including all edges of G that are incident with $m(x)$ if x is a node.

Such a step is denoted as $G \Rightarrow_{m,C,r} G'$. For a finite or infinite set $\mathcal{R}$ of conditional rules, we write $G \Rightarrow_{\mathcal{R}} G'$ if $G \Rightarrow_{m,C,r} G'$ for some match m and some $(C, r) \in \mathcal{R}$. As usual, $\Rightarrow_{\mathcal{R}}^*$ shall denote the reflexive-transitive closure of $\Rightarrow_{\mathcal{R}}$.

The default way of rewriting in GrGen is via injective matches, but a specification `hom(`x`,`y`)` allows that the items x and y in P are identified by a match. This can be modeled by extending the rule set $\mathcal{R}$ by a variant of the rule wherein x and y are identical. However, a potential match $\tilde{m}(\tilde{P})$ of a negative pattern may always overlap with the match $m(P)$ of the pattern in an arbitrary way.

Rules as Graphs. Since rules shall be instantiated by applying other rules to them (in the next section), it is important to note that a conditional rule can be represented as a single graph. The *rule graph* $\langle C, r \rangle$ of a conditional rule (C, r) is

[4] Actually, this is a test for the existence of a cytosine molecule in a graph.




```
1 rule NextChainElem(prev:D, rprev:R):(D,R)
2 {
3   prev -:PG-> d:D;
4
5   modify {
6     rprev -:PG-> r:R;
7     return(d,r);
8   }
9 }
```

Fig. 2. Rule `NextChainElem` extending a ribose chain

obtained from the disjoint union of its graph components $P \uplus R \uplus \biguplus_{(c\colon P \to \tilde{P}) \in C} \tilde{P}$ by identifying, for every $x \in Dom(r)$, x with $r(x)$, and for every $c\colon P \to \tilde{P} \in C$ and every $y \in Dom(c)$, y with $c(y)$.

Example 3 (A Rule Graph). The rule in Figure 2 extends a ribose chain in correspondence to a deoxyribose chain. Here and in the following examples, DNA and RNA are represented by chains, with nodes (of type `D` for deoxyribose and `R` for ribose) representing the sugars, and edges of type `PG` representing the phosphate groups linking them. Nodes labeled `A`, `C`, `G`, `T`, and `U` that are connected to the sugars represent the nucleic bases.

The textual notation of the rule in the specification language of GrGen is shown on the right-hand side. The rule has a name (`NextChainElement`), two node parameters (`prev`, `rprev`) and two result nodes (`d`, `r`). Parameters may be used in the body, and results are indicated by ***`return`***. The outer block defines the pattern of the rule (in line 3); it contains a ***`modify`***-block that specifies how items shall be added to the pattern (in line 6). A ***`delete`***-list could indicate nodes and items to be removed from the pattern; this is not used in our example. One or more ***`negative`***-blocks could define negative application conditions (as in rule `DNAchain` of Example 5).

The rule graph of `NextChainElement` is shown on the left-hand side of the figure. The rule name appears at the top of the graph, its parameters are annotated with $\langle in \rangle$, and its results with $\langle out \rangle$. Items in R that are not in $r(P)$ are annotated with $\langle create \rangle$, whereas items in P that are not in $Dom(r)$ would be annotated with $\langle delete \rangle$, and items of the negative application condition would be crossed out.

3 Controlled Graph Rewriting

Controlled rewriting is typically expressed by operations that combine single rule applications. As an example, we summarize the (*graph*) *rewrite sequences* offered by GrGen.

- A rule application $(y_1, \ldots y_k) = r(x_1, \ldots, x_m)$ attempts to extend the matches of its parameters $x_1, \ldots, x_m$ to an arbitrary match of its pattern so that the rule can be applied. If this is possible, the application *succeeds*, and defines the variables $y_1, \ldots y_k$; otherwise it *fails*. A test is handled the same way, but does not modify the graph.
- For rewrite sequences S_1, S_2, the logical operations conjunction $S_1 \,\&\&\, S_2$ and disjunction $S_1 \,||\, S_2$ are evaluated lazily from left to right: S_2 is not evaluated if the success or failure of S_1 does already determine the result of the operation. Their strict counterparts &, |, and the negation ! exist as well.
- Iteration is supported by the constructs S^* and S^+ which evaluate a rewrite sequence S until it fails. S^* never fails, and S^+ is equivalent to $S \,\&\&\, S^*$.
- Transactional brackets $\langle S \rangle$ undo all effects of intermediate evaluation steps in a rewrite sequence S if the evaluation of S as a whole fails. Backtracking however – in the sense of exploring all possible rewrite sequence applications automatically – is not supported; this yields high efficiency in many cases, but complicates handling of recursive structures, as it is not possible to simply restart a stuck sequence at the last decision point.

A. Habel and D. Plump have shown in [13] that rewrite programs supporting *(i)* choice of one rule from a set of (DPO) graph rewrite rules, *(ii)* sequential composition, and *(iii)* exhaustive application suffice to define every computable function on graphs. However, this does not mean that this kind of control supports practical programming in an optimal way.

Example 4 (A Graph Rewrite Sequence for DNA Transcription). The following graph rewrite sequence performs DNA-to-RNA transcription like the Haskell function in Example 1.

```
1 < (prev,rprev) =  findTATAAA
2  && ( !isCCCACTuvwxyzAGTGGGAAAAA(prev)
3       && (prev,rprev)=NextChainElement(prev,rprev)
4       && (A(prev,rprev) || C(prev,rprev) || G(prev,rprev) || T(prev,rprev))
5     )*
6  &&  isCCCACTuvwxyzAGTGGGAAAAA(prev) >
```

The rule `findTATAAA` searches for the transcription starting sequence, the rule `NextChainElement` known from Example 3 extends the ribose chain to the rear, and the rules `A`, `C`, `G`, and `T` construct the nucleic base pair for the RNA chain that is complementary to the nucleic base in the DNA chain. (Their rules are similar to the alternatives of the pattern `DNANucleotide` shown in Example 5 further below.)

It is remarkable that the rewrite rules themselves perform rather trivial tasks (finding a subsequence, duplicating a chain element, attaching a node), whereas the controlling rewrite sequence that combines them is rather complex, even for such a small example. For efficient rewriting it is important to pass nodes matched in one rule to another one. Then we cannot only control *which* rule is to be applied next, but may also indicate *where* it shall be applied. Rewrite sequences can achieve this only for linear structures like lists, but for non-linear recursive structures like trees, parameter passing cannot be handled without general recursion in rewrite programs. The concepts devised for overcoming the limitations of rewrite programs are described in the next section.

4 Generic Rules

In his master thesis [15], E. Jakumeit has designed and implemented rules with structural recursion. Extending the rules strengthens the rule-based kernel of GrGen, rather than the rewrite sequences defined on top of it. The basic idea is that a generic rule contains variables, nonterminal nodes which are attached to a fixed number of terminal nodes. A set of structure rules describes how variables can be substituted. A variable may have several substitutions, which can be used alternatively. These substitutions may again contain variables, even in a recursive way. If the variable occurs in a pattern (positive or negative) of the generic rule, its instantiation yields a sub-pattern. However, since generic rules are represented as rule graphs, a variable may be attached to its (positive) pattern and replacement at the same time. Then its instantiation yields a sub-rule. Both sub-patterns and sub-rules are defined by structural recursion.

Formally, the semantics of generic rules is defined by a two-level graph rewrite process. First, all variables in a generic rule are instantiated according to the structure rules, by a context-free way of graph rewriting similar to hyperedge replacement [10]. This process yields a language of simple rules that may be infinite. Then, the host graph is rewritten with the resulting simple rules.

Assumption (Nonterminal Types). We assume that the type alphabets $T = (\dot{T}, \bar{T})$ contain a subset $N \subseteq \dot{T}$ of *nonterminals*, which are equipped with an *arity function* $\mathcal{A}\colon N \to \wp(\bar{T} \times (\dot{T} \setminus N))$.

For all graphs G occurring in the following, we assume that nonterminals are used according to their arity: Whenever G contains a node x with $\dot{\tau}_G(x) = n \in N$, G shall contain, for every $(\bar{t}, \dot{t}) \in \mathcal{A}(n)$, exactly one edge e and node y with $src_G(e) = x$ and $tgt_G(e) = y$ so that $\bar{\tau}_G(e) = \bar{t}$ and $\dot{t} = \dot{\tau}_G(y)$.

Nonterminals will occur only during instantiation, as types of variables in generic rules or in structure rules, but neither in the host graphs, nor in the simple rules applied to them. The remaining types, $(\dot{T} \setminus N) \cup \bar{T}$, are called *terminal*, as well as rules and graphs over these types.

Variables. A node x with type $n \in N$ is called a *variable*. A variable x is called *straight* if it has as many incident edges as adjacent nodes. A subgraph S induced by the incident edges of a variable is called a *star*, and its edges are called rays, and drawn like that (see Figure 3).

Structure Rules. A rule $s = S \dashrightarrow \langle r \rangle$ is a *structure rule* if its pattern S is a straight star, $\langle r \rangle$ is the graph of an unconditional rule $r\colon P \dashrightarrow R$, and $Dom(s)$ is the discrete subgraph that contains all terminal nodes of S. With S_p we denote the maximal subgraph of $Dom(s)$ so that its image $s(S_p)$ is in the pattern P of the rule graph $\langle r \rangle$. A structure rule $s = S \dashrightarrow \langle r \rangle$ is a *sub-pattern structure rule* if the morphism $r\colon P \dashrightarrow R$ is total and surjective, otherwise, it is a *sub-rule structure rule*.

Instantiation of Generic Rules. A conditional rule (C, r) with $r\colon P \dashrightarrow R$ is called *generic* if every variable y in R has a variable x in P with $r(x) = y$.

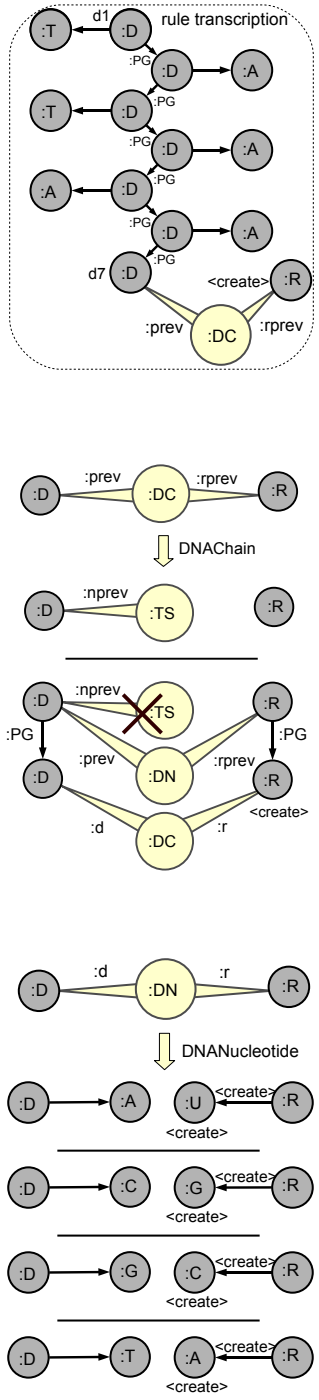
Let $T = \langle C, r \rangle$ be the graph of a generic rule and consider a structure rule $s = S \dashrightarrow \langle \hat{r} \rangle$. A total morphism $m\colon S \to T$ is a *rule match* if $m(S_p)$ is a subgraph of the pattern of the rule graph $\langle C, r \rangle$, or, if s is actually a sub-pattern structure rule (and $S_p = S$), if $m(S)$ either lies completely in a negative pattern $\tilde{P}$ (where $c\colon P \to \tilde{P} \in C$), or in the pattern of $\langle C, r \rangle$. Then $T \Rightarrow_{m,\emptyset,s} T'$ is an *instantiation step*, where the transformed graph is a rule graph $T' = \langle C', r' \rangle$ again, whose negative patterns, pattern and replacement can be distinguished by considering the pushouts for S_p and $S \setminus S_p$ separately.

Let $\mathcal{S}$ be a finite set of structure rules, and define $\Rightarrow_{\mathcal{S}}$ to be its instantiation relation. Then $\mathcal{S}$ derives, for some set $\mathcal{R}$ of generic conditional rules, the set of simple conditional rules

$$\mathcal{S}(\mathcal{R}) = \{\langle C', r' \rangle \mid \langle C, r \rangle \in \mathcal{R}, \langle C, r \rangle \Rightarrow^*_{\mathcal{S}} \langle C', r' \rangle, \text{ where } \langle C', r' \rangle \text{ is terminal}\}$$

The rewrite relation of generic rules $\mathcal{R}$ over structure rules $\mathcal{S}$ is given as $\Rightarrow_{\mathcal{S}(\mathcal{R})}$.

Example 5 (Transcription of DNA to RNA). Coming back to Examples 1 and 4, we show a generic rule transcribing DNA to RNA in Figure 3. Now the transcription can be specified by a single rule, with three nonterminals `DNAChain`, `DNANucleotide`, and `isCCCACTuvwxyzAGTGGGAAAAAA`. We first discuss the textual notation of GrGen on the right-hand side of the figure. Six structure rules define the sub-rules `End`, `Chain` and `A`, `C`, `G`, `T` for the first two nonterminals; `Chain` uses `DNAChain` recursively. The rays and adjacent nodes of these nonterminals are given by the names and types of the formal parameters that follow their name, plus those that follow the `modify` blocks in their rules. The structure rule for the nonterminal `isCCC...AAA` defines a sub-pattern; note that it is used for two (anonymous) variables: as a negative application pattern in the structure rule `End`, and as a positive pattern in `Chain`. Using two (or more) variables of the same structure within one rule is also important for expressing structural recursion over non-linear structures like trees.




```
1  rule transcription() {
2    d1:D -:PG-> d2:D -:PG-> d3:D -:PG-> d4;
3    d4:D -:PG-> d5:D -:PG-> d6:D -:PG-> d7;
4    d7:D; d1 --> :T; d2 --> :A; d3 --> :T;
5    d4 --> :A; d5 --> :A; d6 --> :A;
6    d2r:DNAChain(d7);
7    modify {
8      r:R; // new starting point for RNA
9      d2r(r);
10   }
11 }
12 pattern DNAChain(prev:D) {
13   alternative {
14     End {
15       :isCCCACTuvwxyzAGTGGGAAAAAA(prev);
16       modify(rprev:R) {
17       }
18     }
19     Chain {
20       negative {
21         :isCCCACTuvwxyzAGTGGGAAAAAA(prev);
22       }
23       prev -:PG-> next:D;
24       head:DNANucleotide(prev);
25       tail:DNAChain(next);
26       modify(rprev:R) {
27         rprev -:PG-> rnext:R;
28         head(rprev);
29         tail(rnext);
30       }
31     }
32   }
33   modify(rprev:R) { }
34 }
35 pattern DNANucleotide(d:D) {
36   alternative {
37     A { // analogously for C, G. T
38       d --> a:A;
39       modify(r:R) {
40         r --> u:U;
41       }
42     }
43     ...
44 }
45 pattern isCCCACTuvwxyzAGTGGGAAAAAA(d:D) {
46    ...
47 }
```

Fig. 3. DNA-to-RNA Transcription defined with a generic rule

Note that the abstract DNA model employed here, where nodes and edges represent sub-molecules, can easily be defined on the underlying chemical structure that is composed of atoms. To do that, every nucleotide node (of type `A`, `C`, `G`, `T`, or `U`), every phosphate group edge (of type `PG`), and every sugar node (of type `D` or `R`) has to be turned into a nonterminal, whose structure rules specify the corresponding sub-molecules, and have one, two, and three attachment points respectively, which have to be joined according to the chemical bonds between these sub-molecules. See [15] for details.

Rule Application. Instantiating generic rules first, and matching them afterwards is only possible in theory —in practice, we have to interleave instantiation with matching, as sketched in the following *operational semantics* of the recursive rules, which has been implemented in the extension of GrGen [15]:

1. The terminal items in the generic rule's pattern are matched.
2. It is checked whether the terminal items of a negative pattern may be matched.
3. If this is the case, a variable attached to this negative pattern is substituted according to a structure rule, and matching continues in step 2. If no variable is left in the negative pattern, a match is found, and application of the rule fails.
4. Otherwise, a variable attached to the pattern is substituted with one of its structure rules, and matching continues with step 4. If there is no variable anymore in the pattern, application of the rule succeeds.
5. The replacement of the generic rule—wherein variables are now instantiated—replaces the match of the rule.

The structure rules $\mathcal{S}$ correspond to hyperedge replacement graph grammars. Thus non-productive nonterminals, unused nonterminals, and chain rules can be detected and removed [10]. When we assume $\mathcal{S}$ to be free of such nonterminals and rules, the operational semantics is effective, since the substitution process is bound to terminate. If the rules are defined with care, it can also be efficient.

5 Conclusions

In this paper we have described a concept by which graph rewrite rules can be refined recursively so that advanced transformation tasks can be specified by a single rule, without using imperative control structures. The concept has been implemented in GrGen.[5] Due to lack of space, we have simplified the full concepts of GrGen in several respects: Nodes and edges of graphs may carry attribute values, their typing may use inheritance, and the structure rules used for refining generic rules may themselves be conditional.

[5] A beta version of GrGen.NET 2.0 is available at www.grgen.net

The idea of using rules to refine rules has been first used in two-level (van-Wijngaarden) grammars [3]. Early adaptations of this idea to graph grammars [14, 9] were oriented towards defining graph languages, and not intended for defining computations on graphs. The graph variables of shaped generic graph rewrite rules [5] resemble the variables introduced here; they are refined by adaptive star replacement [4]. This is more general than the star replacement used here, but more difficult (and less efficient) to implement. Graph variables as such were first proposed in [18], but without the capability to constrain the shape of the graph to be matched. The path expressions and multi-nodes of PROGRES [19] and FUJABA [7] allow matching of a subset of the structures which can be handled by recursive sub-patterns. (In contrast to the instantiations defined here, the match of a path expression may overlap with the rest of a match.) FUJABA [7] as well as earlier versions of GRGEN furthermore support recursion on the right hand side of a rule, i.e., it is possible to call a rule during a rewrite step, after the match is done. This is purely imperative, because the calling rule will make its changes to the graph anyway—regardless whether the called rule is applicable or not. VIATRA [1] was the first graph rewrite system to support recursive sub-*patterns*, sub-*rules* however are not supported (they are only vaguely sketched in the given reference). As of now it still is the only other system offering sub-patterns, but about two orders of magnitude slower than GRGEN [15]. In any of the mentioned cases, variables are placeholders for sub-patterns only, so that recursive patterns can get matched, but not rewritten (besides deleting the entire sub-pattern).

An interesting question for the future is: *Can rules and patterns be merged to a single concept?* Then, generic rules could refer to other rules like to variables, and the application of a rule could "call" other rules, also recursively. With an additional concept for the sequential composition of rules, this could set up a fully declarative way of programming with graph rewrite rules that is computationally complete in the sense of [13]. For such a declarative framework, it is also promising to analyze properties of generic rules, such as the existence of critical pairs, and to try to transfer first results concerning overlapping rules with graph variables [12] to this framework.

References

1. András Balogh and Dániel Varró. Pattern composition in graph transformation rules. In *European Workshop on Composition of Model Transformations*, Bilbao, Spain, July 2006. See also http://viatra.inf.mit.bme.hu/update/R2.
2. Jakob Blomer and Rubino Geiß. GRGEN.NET: A generative system for graph-rewriting, user manual. www.grgen.net, 2007.
3. C.J. Cleaveland and R.C. Uzgalis. *Grammars for Programming Languages*. Elsevier, New York, 1977.
4. Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Adaptive star grammars. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *3rd Int'l Conference*

on Graph Transformation (ICGT'06), number 4178 in Lecture Notes in Computer Science, pages 77–91. Springer, 2006.
5. Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Shaped generic graph transformation. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, Lecture Notes in Computer Science. Springer, 2008. to appear.
6. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in Lecture Notes in Computer Science. Springer, 2000.
7. Thorsten Fischer, Jörg Niere, Lars Turunski, and Albert Zündorf. Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java. In Ehrig et al. [6], pages 296–309. http://www.fujaba.de/.
8. Rubino Geiß. *Graphersetzung mit Anwendungen im Übersetzerbau (in German)*. Dissertation, Universität Karlsruhe, 2007.
9. Herbert Göttler. Semantical descriptions by two-level gaph-grammars for quasi-hierarchical graphs. In Manfred Nagl and Hans-Jürgen Schneider, editors, *Graphs, Data Structures, Algorithms (WG'79)*, number 13 in Applied Computer Science, pages 207–225, München-Wien, 1979. Carl-Hanser Verlag.
10. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Number 643 in Lecture Notes in Computer Science. Springer, 1992.
11. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26:287–313, 1996.
12. Annegret Habel and Berthold Hoffmann. Parallel independence in hierarchical graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *2nd Int'l Conference on Graph Transformation (ICGT'04)*, number 3256 in Lecture Notes in Computer Science, pages 178–193. Springer, 2004.
13. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.
14. Wolfgang Hesse. Two-level graph grammars. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 255–269. Springer, 1979.
15. Edgar Jakumeit. *Mit* GrGen *zu den Sternen*. Diplomarbeit (in German), Universität Karlsruhe, 2008.
16. Sabine Kuske. More about control conditions for transformation units. In Ehrig et al. [6], pages 323–337.
17. Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.
18. Detlef Plump and Annegret Habel. Graph unification and matching. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
19. Andy Schürr, Andreas Winter, and Albert Zündorf. The Progres approach: Language and environment. In Gregor Engels, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, chapter 13, pages 487–550. World Scientific, Singapore, 1999.

Assuring Strong Termination of Controlled Graph Transformation by Means of Petri Nets*

Renate Klempien-Hinrichs and Melanie Luderer

Department of Computer Science, University of Bremen, Germany
{rena,melu}@informatik.uni-bremen.de

Abstract. Termination is an important problem for graph transformation systems, but in general it is undecidable. In this paper we propose an algorithm that searches for sufficient conditions to ensure termination of graph transformation controlled by regular expressions. The main idea is to make use of the recursive structure of regular expressions and compute for each considered expression a finite set of upper bounds with respect to what is deleted and added by derivations permitted by the expression. Elements of this set may interact when iterating the expression. This is analysed by means of a Petri net: Non-repetitiveness of the net implies termination of the considered control expression. Finally, we show that the findings can be transferred to regular expressions extended by *as-long-as-possible*.

1 Termination for Graph Transformation Systems

In rule-based graph transformation (see [Roz97,EEKR99,EKMR99] for an overview), graphs are transformed step-by-step through applications of rules that usually come from a finite set. When using graph transformation as a programming paradigm, termination is an important issue, which is studied, e.g., in [Plu98,Aßm00,BHPPT05,EEdL+05,VVGE+06,LPE07,HKK08]. In general, termination is undecidable for graph rewriting systems [Plu98]. But for many systems termination can be guaranteed. A sufficient criterion is to find a termination function, i.e. an evaluation function $eval\colon \mathcal{G} \to \mathbb{N}$ that associates a natural number $eval(G)$ with each graph $G \in \mathcal{G}$ such that the value decreases whenever a derivation step is done: $eval(G) > eval(G')$ for $G \Longrightarrow G'$. More generally, one may replace $\mathbb{N}$ by some ordered domain which does not have any infinite decreasing sequence (cf. [DM79]). However, requiring a single termination function to work for *every* rule in a system is very restrictive, at least from a practical point of view.

*This research was partially supported by the International Graduate School for Dynamics in Logistics at the University of Bremen and the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

For many applications of graph transformation, arbitrary rule application sequences are undesirable. Rather, one wishes to select sequences that satisfy so-called control conditions. A control condition may, e.g., express a property of a whole rule application sequence (hence 'condition'), or reduce in each derivation step the number of choices for the next rule. Typical conditions are regular expressions over rules (and imported transformation units [KK99,Kus00b]), regular expressions together with *as-long-as-possible*, and priorities (see, e.g., [Kus00a,HKK08]). Layers of rules, for which termination is studied in [EEdL$^+$05], may be seen as a special kind of priorities. In [BHPPT05], the concept of termination function is developed for expressions with *as-long-as-possible*.

The research on which we report in this paper started with the observation that a derivation controlled by applying an expression C_0 *as-long-as-possible* is infinite, i.e. does not terminate, only if C_0 itself admits a non-terminating derivation or C_0^* admits infinitely many derivations (of finite length). Therefore, we first study regular expressions over rules as control conditions and propose a notion of *strong termination* for such expressions, meaning that only finitely many derivations are admitted. Then we show that our approach transfers easily to regular expressions with *as-long-as-possible*.

Moreover, the approach taken in this paper is constructive in the sense that given a finite set of (simple) measures for graphs and a regular expression C, we develop a method to search for a subset of the measures that allows us to claim strong termination of C, even without explicitly stating a termination function. For this, we make use of the recursive structure of regular expressions, and assume worst-case situations for starred subexpressions for which we then construct a Petri net whose behaviour may ensure strong termination of the starred subexpression. (A similar idea to exploit Petri net properties, but for an encoding of single rules, has been explored in [VVGE$^+$06].)

In the tradition of transformation units [KK99,Kus00b], our considerations are independent of any specific graph transformation approach such as, e.g., node rewriting, double-pushout, single-pushout, or high-level replacement. Consequently, they hold for all approaches satisfying some basic requirements.

The paper is structured as follows. In Section 2 we recollect the notions of graph transformation approach and regular expressions as control conditions and summarize basics from Petri net theory. Our method is developed in Section 3. A summary and some ideas for future work conclude the paper.

2 Preliminaries

$\mathbb{N} = \{0, 1, 2, \dots\}$ denotes the set of natural numbers. Let $[n] = \{1, 2, \dots n\}$ for $n \in \mathbb{N}$. $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$ is the set of integers.

Graph transformation approach. There are various kinds of graphs (node- and/or edge-labelled, simple or with parallel edges, edges or hyperedges, etc.) and

ways how to transform them, see [Roz97] for an overview. The considerations in this paper are independent of a specific graph transformation approach; therefore we have to define some basic requirements for such an approach. Compared with the usual definition, we leave out graph class expressions (used to define classes of initial and terminal graphs) and identifiers (that refer to rules or graph transformation units). Moreover, we consider only control conditions that are explicitly based on rules.

A *graph transformation approach* is a system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Longrightarrow, \mathcal{C})$ where $\mathcal{G}$ is a class of *graphs*, $\mathcal{R}$ is a class of *rules*, $\underset{r}{\Longrightarrow}$ is the derivation relation associated with $r \in \mathcal{R}$, and $\mathcal{C}$ is a class of *control conditions* over $\mathcal{R}$.

If the application of a rule $r \in \mathcal{R}$ to a graph G yields the graph G' we write $G \underset{r}{\Longrightarrow} G'$ and call this a *derivation step*. A sequence of rule applications $G_0 \underset{r_1}{\Longrightarrow} G_1 \underset{r_2}{\Longrightarrow} \ldots \underset{r_n}{\Longrightarrow} G_n$ is called a *derivation* and may also be denoted by $G_0 \underset{r_1 \cdots r_n}{\Longrightarrow} G_n$. In this case, we call the rule sequence $r_1 \cdots r_n$ a *rule application sequence* (for G_0). Moreover, if $r_1, \ldots, r_n \in P$ for some set $P \subseteq \mathcal{R}$, we may write the derivation as $G_0 \overset{n}{\underset{P}{\Longrightarrow}} G_n$ or $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$. For a derivation $d = (G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n)$ we write $\mathit{start}(d) = G_0$ and $\mathit{end}(d) = G_n$.

For the rest of this paper, we will assume the following:

Assumption 1. 1. For each rule $r \in \mathcal{R}$, there is at least one graph $G \in \mathcal{G}$ to which it can be applied, i.e. its derivation relation $\underset{r}{\Longrightarrow}$ is not empty.
2. Each rule $r \in \mathcal{R}$ can be applied to a graph $G \in \mathcal{G}$ only in finitely many ways, i.e. the set $\mathit{der}(G)_r = \{G \underset{r}{\Longrightarrow} G' \mid G' \in \mathcal{G}\}$ is finite.

Control Conditions. A major purpose of control conditions is to regulate the derivation process by enforcing some kind of order on rule applications, thus reducing – but in general not eliminating – the non-determinism inherent in graph transformation. Various kinds of control conditions and the interrelations between them are studied in [Kus00a,Kus00b]. A very natural kind is a regular expression over $\mathcal{R}$, which denotes a regular language of strings over $\mathcal{R}$. A derivation satisfies this condition if its rule application sequence is a string in the regular language. Moreover, regular expressions are often extended with an operation *as-long-as-possible*.

The set $\mathit{REX}(\mathcal{R})$ of regular expressions over $\mathcal{R}$ is defined as usual: $\emptyset$ and λ are regular expressions, each $r \in \mathcal{R}$ is a regular expression, and $(e_1; e_2)$, $(e_1|e_2)$, (e^*) are regular expressions for all regular expressions e_1, e_2, e. A regular expression is *star-free* if it does not contain a subexpression of the form (e^*).

Every regular expression e defines a set $L(e) \subseteq \mathcal{R}^*$ as usual: $L(\emptyset) = \emptyset$, $L(\lambda) = \{\lambda\}$, $L(r) = \{r\}$ for $r \in \mathcal{R}$, $L(e_1; e_2) = L(e_1)L(e_2)$ and $L(e_1|e_2) = L(e_1) \cup L(e_2)$ for regular expressions e_1, e_2, and $L(e^*) = L(e)^*$ for a regular expression e.

Let $G \in \mathcal{G}$ be a graph. For a rule sequence w over $\mathcal{R}$, the set of derivations permitted by w (w-*runs* for short) starting in G is defined as $\mathit{der}(w)_G =$

$\{G \underset{w}{\Longrightarrow} G' \mid G' \in \mathcal{G}\}$. For a regular expression e over $\mathcal{R}$, the set of permitted derivations (*e-runs* for short) starting in G is $der(e)_G = \bigcup_{w \in L(e)} der(w)_G$.

When programming with graph transformation, one usually has no need to specify the empty set of derivations or the set containing only the empty derivation. Therefore, we will from now on ignore regular expressions $\emptyset$ and λ.

The set of regular expressions is extended by the operation *as-long-as-possible*, denoted by !, to a set of expressions as follows: $(e!)$ is an expression if e is an expression. The meaning of $(e!)$ is to iterate derivations permitted by e as often as a complete derivation permitted by e can be executed.

Clearly, the number of iterations depends on the chosen start graph and may even be infinite. Therefore, we cannot associate a language $L(e!) \subseteq \mathcal{R}^*$ with $(e!)$ that is valid for all start graphs.

The set of permitted derivations $der(e!)_G$ contains all finite concatenations $d_1 d_2 \cdots d_n$ of derivations $d_1 \in der(e)_G$ and $d_{i+1} \in der(e)_{end(d_i)}$ for $i \in [n-1]$ so that $der(e)_{end(d_n)} = \emptyset$, and all infinite concatenations $d_1 d_2 \ldots$ built analogously if $der(e)_{end(d_i)} \neq \emptyset$ for all $i > 0$.

Petri Nets. Petri nets are a well-known modelling tool for nondeterministic concurrent systems. A concise introduction can be found in, e.g., [Mur89].

A *Petri net* is a system $N = (P, T, F, B)$ where P is a finite set of *places*, T is a finite set of *transitions* with $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$, and F, B are $|P| \times |T|$-matrices over $\mathbb{Z}$ called *forward matrix* and *backward matrix*, respectively. These matrices give rise to the *incidence matrix* $A = B - F$, which contains already all information about N if N is *pure*, i.e. there is no place $p \in P$ and transition $t \in T$ with both $F(p,t) > 0$ and $B(p,t) > 0$.

Any mapping $M \colon P \to \mathbb{N}$ is called a *marking* of N. The Petri net N with *initial marking* M_0 is denoted by $N(M_0) = (P, T, F, B, M_0)$. Transition $t \in T$ is *enabled* at marking $M \colon P \to \mathbb{N}$ if $F(t) \leq M$, i.e. $F(p,t) \leq M(p)$ for all places $p \in P$. Then t may *fire* to marking $M' = M + A(t)$, which is denoted by $M\,[t\!>\, M'$. A transition sequence $t_1 \cdots t_n \in T^*$ is a *firing sequence* starting from marking M, denoted by $M\,[t_1 \cdots t_n\!>\, M'$, if there are markings $M_0, M_1, \ldots, M_n$ such that $M = M_0$, $M_{i-1}\,[t_i\!>\, M_i$ for all $i \in [n]$, and $M_n = M'$.

A Petri net is *partially repetitive* if there exists a marking M_0 and an infinite firing sequence $w = (t_i)_{i \in \mathbb{N}}$ starting from M_0. Partial repetitivity can be characterised with the help of the incidence matrix (see [Mur89, Theorem 31]).

Theorem 2. *Let $N = (P, T, F, B)$ be a Petri net and A its incidence matrix. N is partially repetitive if and only if there exists a $|T|$-vector $x \colon T \to \mathbb{Z}$ of non-negative integers such that $A \cdot x \geqslant 0$ and $x \neq 0$.*

3 Computing a Sufficient Condition for Termination

The section starts with the basis for a small running example, before a notion of strong termination is defined for regular expressions. Subsequently, measures for graphs and upper bounds for the changes in measures through derivations controlled by regular expressions are defined. On this basis, Petri nets are constructed whose analysis may yield a sufficient condition for the strong termination of a starred expression. Finally, the reflections of this section are condensed into a checking algorithm, and expanded to deal with control conditions that are regular expressions over rules together with *as-long-as-possible*.

Running Example. In this example, let symbol a denote a graph $\circ\!\!\overset{a}{\!\!-\!\!-\!\!-}\!\!\circ$, and several symbols the disjoint union of such graphs. Consider the rules $r_1 : a \to bb$, $r_2 : bb \to c$, $r_3 : a \to d$, $r_4 : a \to aa$, $r_5 : ccc \to d$, which are applied to a graph by locating the left-hand side in the graph, removing it, and adding the right-hand side, so that they may be seen as double-pushout rules with empty interface graph. Moreover, consider control conditions $C_1 = r_1$, $C_2 = (r_2; (r_3^*))$, $C_3 = (r_4; r_5)$ and $C_0 = ((C_1 \mid C_2) \mid C_3)$. Then there are, for instance, three derivations starting in the graph $aabb$ and permitted by C_2, yielding the graphs aac, acd, and cdd, respectively.

Termination of Regular Expressions. In any computation model, a computation does not terminate if some loop construct – such as *do X as-long-as-possible* – requires infinitely many computation steps. A necessary condition for this is that X may be repeated infinitely often from some start configuration. If X is controlled by some regular expression C and starts in a graph G, this means that the regular expression C^* admits infinitely many derivations starting in G. This is exactly the situation that we forbid in our notion of *strong termination* for regular expressions. As a nice consequence, any exhaustive search of derivations admitted by some regular expression will terminate (in the usual sense) if the expression terminates strongly.

Observe that C_1, C_2, C_3 and C_0 from the running example terminate strongly.

Definition 3. Let C be a regular expression over $\mathcal{R}$ and $G \in \mathcal{G}$ a graph.

1. *C terminates strongly for G* if $der(C)_G$ is finite.
2. *C terminates strongly* if it terminates strongly for all graphs in $\mathcal{G}$.

Due to Assumption 1, we have the following observation, where point 2 is just point 1 rephrased since for a regular expression C that contains neither $\emptyset$ nor λ, the language $L(C)$ is finite if and only if C is star-free.

Observation 4. *Let C be a regular expression over $\mathcal{R}$.*

1. *If $L(C)$ is finite then C terminates strongly.*

2. *If C is star-free then C terminates strongly.*
3. *$C_1; C_2$ terminates strongly if and only if C_1 terminates strongly and for all $G \in \mathcal{G}$ and $d \in der(C_1)_G$, C_2 terminates strongly for $end(d)$.*
4. *$C_1|C_2$ terminates strongly if and only if C_1 and C_2 terminate strongly.*
5. *If C^* terminates strongly then C terminates strongly.*

Lemma 5. *Let C be a regular expression over $\mathcal{R}$.*

1. *If $C = r \in \mathcal{R}$ then C terminates strongly.*
2. *If $C = C_1; C_2$ or $C = C_1|C_2$ and both C_1 and C_2 terminate strongly then C terminates strongly.*

It would be nice to have the implication 'if $C = C_0^*$ and C_0 terminates strongly then C terminates strongly,' too. Unfortunately, it is in general false: if C_0 is a rule that just adds a node to any graph then C_0 terminates strongly, but C does not terminate strongly. Still, C_0^* will terminate strongly if for every start graph an intermediate graph is reached after some C_0-runs so that C_0 cannot be applied anymore. This case is investigated in the following sections.

Measure Sets for Graphs. A measure maps graphs to natural numbers such that for each rule, its application yields the same change in the measured value, independently of the graph to which the rule is applied.

For the running example, we consider μ_a, μ_b, μ_c, μ_d as as measures, counting the occurrences of the respective symbol a, b, c, d in a graph.

Definition 6. A *measure* on graphs is a mapping $\mu\colon \mathcal{G} \to \mathbb{N}$ such that for all graphs $G, G', \bar{G}, \bar{G}' \in \mathcal{G}$ and every rule $r \in \mathcal{R}$, $G \underset{r}{\Longrightarrow} G'$ and $\bar{G} \underset{r}{\Longrightarrow} \bar{G}'$ implies $\mu(G') - \mu(G) = \mu(\bar{G}') - \mu(\bar{G})$. We will write a set of k measures $\mu_1, \ldots, \mu_k$ ($k \in \mathbb{N}$) as a vector $\vec{\mu} = (\mu_i)_{i \in [k]}$.

Possible examples for measures are (cf. [Aßm00]): number of nodes, number of edges, number of a-labelled edges or number of b-labelled loops (for a, b symbols of the graph-labelling alphabet). Of course, which mappings qualify as measures depends on the graph transformation approach and the rules occurring in the considered control condition. For instance, node-rewriting rules (see, e.g., [ER97]) usually admit implicit multiplication of embedding edges, so that an edge-based mapping is no measure. If, however, every rule in the concrete set will just transfer every incident a-labelled edge to exactly one replacing node, counting a-labelled edges is valid as a measure.

Upper Bounds for C. Define a set $\mathit{Change}(C) \subseteq \mathbb{Z}_\infty^k$ of vectors for each regular expression C so that each vector has k entries in $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$ that serve as upper bounds for the change in measured values whenever a derivation admitted by C is executed.

Definition 7. Let C be a regular expression over $\mathcal{R}$ and $\vec{\mu}$ a measure set.

1. For $C = r \in \mathcal{R}$, let $\mathit{Change}(C) = \{x\}$, where x is the unique vector $x = \vec{\mu}(G') - \vec{\mu}(G)$ for all $G, G' \in \mathcal{G}$ with $G \underset{r}{\Longrightarrow} G'$, and vector difference is computed component-wise.
2. For $C = C_1; C_2$, let $\mathit{Change}(C) = \{x + y \mid x \in \mathit{Change}(C_1),\ y \in \mathit{Change}(C_2)\}$, where vector addition is computed component-wise.
3. For $C = C_1 | C_2$, let $\mathit{Change}(C) = \mathit{Change}(C_1) \cup \mathit{Change}(C_2)$.
4. For $C = C_0^*$, let $\mathit{Change}(C) = \{(x_1, \ldots, x_k) \mid x_i = \infty$ if $\exists (y_1, \ldots, y_k) \in \mathit{Change}(C_0) : y_i > 0$ and $x_i = 0$ otherwise, for $i \in [k]\}$.

For an expression C_0 that terminates strongly, C_0^* admits rule application sequences where C_0 is iterated arbitrarily often. Any decrease in a measure through a C_0-run does not occur if C_0 is iterated zero times. In contrast, an increase in a measure may lead to arbitrarily large values of that measure, indicated by ∞.

For the three control conditions C_1, C_2, C_3 from the running example, we have sets containing, respectively, the vectors $(-1, 2, 0, 0)^\top$, $(0, -2, 1, \infty)^\top$, $(1, 0, -3, 1)^\top$, and all three vectors form the set $\mathit{Change}(C_0)$.

Lemma 8. *Let C be a regular expression over $\mathcal{R}$.*

1. *For all $G, G' \in \mathcal{G}$ and $w \in L(C)$ with $G \underset{w}{\Longrightarrow} G'$ there exists $x \in \mathit{Change}(C)$ with $\vec{\mu}(G') - \vec{\mu}(G) \leq x$.*
2. *$\mathit{Change}(C)$ is finite.*

Constructing a Petri Net from $\mathit{Change}(C_0)$. Now we are aiming to provide a sufficient condition for strong termination of a regular expression $C = C_0^*$, i.e. for the case that is missing from Lemma 5. Since a necessary condition is the strong termination of C_0 (Observation 4 item 5), we will assume this to be true (or, more precisely, inductively proved) in the following considerations.

Definition 9. Let C_0 be a regular expression over $\mathcal{R}$ that terminates strongly, and let $\vec{\mu} = (\mu_i)_{i \in [k]}$ be a measure set. Construct the pure Petri net N_{C_0} as follows.

- The set of places is $P_{C_0} = \{l \in [k] \mid x_l \neq \infty$ for all $(x_1, \ldots, x_k) \in \mathit{Change}(C_0)\}$,
- the set of transitions is $T_{C_0} = \mathit{Change}(C_0)$,
- the incidence matrix A_{C_0} has as columns the vectors in $\mathit{Change}(C_0)$ restricted to the entries selected for P_{C_0}.

For the running example, the Petri net and matrix for C_0 are given in Fig. 1.

N_{C_0} is well defined since P_{C_0} and T_{C_0} are finite and N_{C_0} is required to be pure. A marking of N_{C_0} may be interpreted as $\vec{\mu}(G)$ restricted to P_{C_0}, for some

$$A_{C_0} = \begin{pmatrix} -1 & 0 & 1 \\ 2 & -2 & 0 \\ 0 & 1 & -3 \end{pmatrix} \qquad N_{C_0} =$$

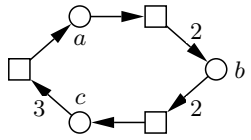


Fig. 1. The incidence matrix A_{C_0} and Petri net N_{C_0} for the running example

graph G. Then the marking reached by firing some transition x records the change from $\overrightarrow{\mu}(G)$ to $\overrightarrow{\mu}(G')$ on the measures retained by P_{C_0} if G' is obtained from G by a derivation whose corresponding vector is x.

Since C_0 is assumed to terminate strongly, we now know the following: If $C = C_0^*$ does not terminate strongly then there is a marking M of N_{C_0} and an infinite firing sequence starting in M, i.e. N_{C_0} is partially repetitive. By contraposition and using Theorem 2 we get the desired sufficient condition for strong termination of C.

Lemma 10. *Let $C = C_0^*$ be a regular expression over $\mathcal{R}$. If C_0 terminates strongly and there is no vector $x\colon T_{C_0} \to \mathbb{Z}$ of non-negative integers such that $A_{C_0} \cdot x \geq 0$ and $x \neq 0$, then C terminates strongly.*

For the running example, we may infer that the only solution for $A_{C_0} \cdot x \geq 0$ is $x = 0$, which implies that $C = (C_0^*)$ terminates strongly.

If N_{C_0} is not partially repetitive then it must be *structurally bounded*, which implies that there exists a vector $y\colon P_{C_0} \to \mathbb{Z}$ of positive integers such that $y \cdot A_{C_0} \leq 0$ (cf. [Mur89, Theorem 29]). If there is even such a vector with $y \cdot A_{C_0} < 0$, then we can show that $y \cdot \overrightarrow{\mu}|_{P_{C_0}}$ is a termination function in the usual sense, where $\overrightarrow{\mu}|_{P_{C_0}}$ denotes the restriction of $\overrightarrow{\mu}$ to P_{C_0}.

A Termination Check. Putting Lemmas 5 and 10 together, we obtain the following test, where the function NoNnSolution returns **true** for an input matrix A over integers if and only if the only non-negative solution x for $A \cdot x \geq 0$ is $x = 0$:

Check(C : regular expression over $\mathcal{R}$) : {**true**, **false**};
case $C \in \mathcal{R}$:
    return true;
  $C = C_1; C_2$ **or** $C = C_1 | C_2$:
    return Check(C_1) **and** Check(C_2);
  $C = C_0^*$:
    return Check(C_0) **and** NoNnSolution(A_{C_0})
endcase

Theorem 11. *Let C be a regular expression over $\mathcal{R}$. If* Check(C) = **true** *then C terminates strongly.*

Regular Expressions with *as-long-as-possible*. Let us consider the extension of regular expressions over $\mathcal{R}$ to include the operation *as-long-as-possible*. Then an expression C *terminates strongly* if for all graphs $G \in \mathcal{G}$, $der(C)_G$ is finite and contains only derivations of finite length. By induction on the structure of nested *as-long-as-possible*'s we can show the following, where the mapping *rex* turns an expression C into a regular expression $rex(C)$ by replacing every occurrence of '!' with '*':

Lemma 12. *Let C be an expression. Then C and all its subexpressions terminate strongly if and only if $rex(C)$ and all its subexpressions terminate strongly.*

Consequently, the algorithm above can be used to search for a sufficient condition for expressions with *as-long-as-possible*, too.

Theorem 13. *Let C be an expression over $\mathcal{R}$. If* CHECK$(rex(C)) =$ **true** *then C terminates strongly.*

In [BHPPT05] an iterated (by *as-long-as-possible*) control condition is supposed to be applicable at least once. Translated into regular expressions, this corresponds to requiring that in a starred subexpression C_0^*, the expression C_0 must be executed at least once. Thus, only subexpressions of the form $C_0^+ = C_0; C_0^*$ would be admitted. For our algorithm, this restriction implies that in computing $Change(C_0^+)$ (cf. Definition 7 item 4) one should take the component-wise maximum of all vectors in $Change(C_0)$. Thus one can possibly maintain some negative entries and gain more positive CHECK results.

4 Conclusion

In this paper we have proposed an algorithm to search for sufficient conditions that ensure (strong) termination of graph transformation processes restricted by a control condition in the form of a regular expression. Strong termination for a regular expression C follows from strong termination for all subexpressions C_0^* of C. The basic idea is that, given strong termination of C_0, C_0^* terminates strongly if every derivation permitted by C_0 deletes something, and during iteration of C_0 at least one type of deleted elements cannot be balanced by other derivations permitted by C_0 that add elements of the same type.

The algorithm needs a finite set of measures for graphs. Then, for each considered regular expression C_0, it constructs a finite set $Change(C_0)$ of upper bounds for the differences of measured elements resulting from a derivation permitted by that expression. This set is interpreted as the incidence matrix of a pure Petri net. If linear-algebraic analysis shows the Petri net to be non-repetitive, strong termination of C_0^* is implied. Moreover, we have shown that regular expressions extended by *as-long-as-possible* can be treated analogously.

It should be noted that the algorithm yields only a sufficient condition for (strong) termination of an input expression. If it returns **false**, it is open whether

the input expression terminates (strongly). While in principle we cannot expect more since termination is undecidable for graph transformation systems [Plu98], there may be room for improvement. For instance, the vectors in $Change(C_0^*)$ have as entries only 0 or ∞. Is there a better treatment for this case?

Ideally, for a transformation unit in the sense of [KK99,Kus00b], one would like to have that the control condition terminates (strongly) for all initial graphs. Our approach may help in three aspects (subject to further study): (1) For a given initial graph, test prior to executing a derivation whether all derivations starting in that graph and admitted by the control condition will terminate strongly. (2) Test whether for all initial graphs the control condition will terminate strongly. (3) Compute from a control condition a set of initial graphs for which that condition terminates strongly.

Acknowledgment. We are grateful to the referees for their thoughtful remarks, which helped to improve the paper.

References

[Aßm00] Uwe Aßmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, 2000.

[BHPPT05] Paolo Bottoni, Kathrin Hoffmann, Francesco Parisi-Presicce, and Gabriele Taentzer. High-level replacement units and their termination properties. *J. Vis. Lang. Comput.*, 16(6):485–507, 2005.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[EEdL+05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *Proc. FASE 2005*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

[EKMR99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [Roz97], pages 1–94.

[HKK08] Karsten Hölscher, Renate Klempien-Hinrichs, and Peter Knirsch. Undecidable control conditions in graph transformation units. *Electron. Notes Theor. Comput. Sci.*, 195:95–111, 2008.

[KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig et al. [EEKR99], pages 607–638.

[Kus00a] Sabine Kuske. More about control conditions for transformation units. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 323–337, 2000.

[Kus00b] Sabine Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.

[LPE07] Tihamér Levendovszky, Ulrike Prange, and Hartmut Ehrig. Termination criteria for DPO transformations with injective matches. *Electr. Notes Theor. Comput. Sci.*, 175(4):87–100, 2007.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4), 1989.

[Plu98] Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[VVGE+06] Dániel Varró, Szilvia Varró-Gyapay, Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Termination analysis of model transformations by Petri nets. In *Proc. ICGT 2006*, volume 4178 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2006.

Efficient Graph Rewriting System Using Local Event-driven Pattern Matching *

Bilal Said and Olivier Gasquet

Université Paul Sabatier, IRIT - LILaC,
118 route de Narbonne, F-31062 Toulouse cedex 9, France

Abstract. Pattern matching is the most time-cost expensive part of the graph rewriting process. When applying a given rewriting rule, most graph rewriting systems check the entire host graph for possible matches. In our modal logic theorem prover based on graph rewriting, LoTREC, we use the events that are emitted when the applied rules change locally the host graph, in order to reduce the effect of this problem by performing the match process on the relevant set of subgraphs.

Introduction

The large variety of graph transformation tools have mainly the same one-step rule application mechanism and usually differ by the techniques used for the graph pattern matching step, which is considered to be the most crucial in the overall performance of a graph transformation process. In fact, a naive implementation computes every possible mapping of the L nodes of the left-hand side of a rewriting rule to N nodes of the host graph, leading to $O(|N^L|)$ time complexity in general.

The classical approach, used in [4] for AGG [6], is to solve the pattern matching problem as a *constraint satisfaction problem*. Other systems, such as PROGRES [8]) and FUJABA [3], use *local search* techniques consisting of matching a single node by some heuristics and extending the matching step-by-step by neighboring nodes and edges. G. Varró and D. Varró introduce in [7] the *incremental update* technique that aims to keep track of all possible matchings identified by graph transformation rules in database tables, and update these tables incrementally to exploit the fact that rules typically perform only local modifications to graphs.

In the graph rewriting system of our theorem prover LoTREC, we implement an original technique that lies between these three approaches. Before running the rewriting process, we analyse the left-hand side conditions of every rule in order to precise the possible ways (i.e. search plans) a matching process could be established and achieved. During the rewriting process, we keep track of the

*This work has been partially supported by the project ARROWS of the French *Agence Nationale de la Recherche*

local changes made by the rules at each step, using a special data structure called *events*. When it is called by the strategy, a rule uses these events to establish a local pattern matching process, with respect to its different search plans. Finally, the established match is completed using a CSP like procedure that instantiates the rest of the pattern graph variables.

It is clear that this technique does not reduce the complexity of the matching process itself. Nevertheless, the pattern matching time-cost becomes $O(|k^L|)$ in the applications, where the rewriting rules are applied on at most k subgraphs at each step.

In section 1 we define our graph rewriting system. Then we present our event-driven pattern matching process in section 2. Finally we sum up with a discussion of our results.

1 The Graph Rewriting System of LoTREC

LoTREC is a theorem prover by tableau for modal logics [1], [2]. It allows the implementation of new user-defined logics, and it is capable of analysing a given input formula and building the models that verify it, and/or the counter-models that refute it. The key similarity with graph rewriting systems is that a (*Kripke*) model is very similar to attributed directed graphs, with formulas being the attributes for nodes and edges, and that tableau rules and their application are to models what rewriting rules and their application are to graphs.

In this paper, we simplify the definition of these graphs to the following:

Definition 1 (Graph). A graph G is a tuple (V, E, F, s, t, f) where V is a finite set of nodes (or vertices) and E a finite set of arcs (or edges), $V \cap E = \emptyset$; s and t are two total mappings $s, t : E \rightarrow V$, denoting "source" and "target"; and F is a set of formulas labelling nodes and edges w.r.t. $f : V \cup E \rightarrow 2^F$.

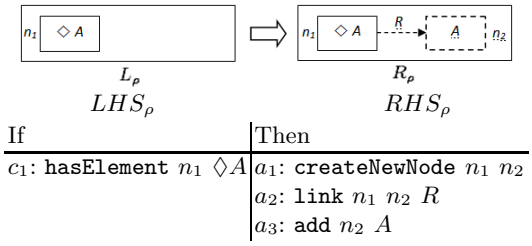
We use $S_G = V_G \cup E_G \cup F_G$ to denote the graph objects (symbols) of G, and $\mu_G \in \{s_G, t_G, f_G\}$ to designate one of the mapping functions of G.

Definition 2 (Graph morphism). A graph morphism between two directed graphs L and G is a total mapping $\mathcal{M} : S_L \rightarrow S_G$ where the total mappings of L and G are preserved, i.e. $\forall s \in S_L : \mathcal{M}(\mu_L(s)) = \mu_G(\mathcal{M}(s))$. Note that $\mathcal{M}(S) = \{\mathcal{M}(s),\ s \in S\}$.

Rule definition Using the Single Push Out (SPO) approach, a rewriting rule ρ is a single graph morphism that maps a *graph pattern* L_ρ, and a *replacement* graph R_ρ.

We define L_ρ and R_ρ using an appropriate simple declarative language (figure 1). In this language, we use two sets of symbols: $\mathcal{T}$, a set of *terms* (or *variable* symbols), and $\mathcal{C}$, a set of *constant* symbols. The graph objects of L_ρ and R_ρ are represented by symbols from a set $S_\rho = T_\rho \cup C_\rho$, where $T_\rho \subseteq \mathcal{T}$, and $C_\rho \subseteq \mathcal{C}$.

We denote by S_{L_ρ} and S_{R_ρ} the subsets of S_ρ appearing in L_ρ and R_ρ. We also use $\mathcal{K}^n$ to denote the set of first order predicates of arity n defined over $\mathcal{T} \cup \mathcal{C}$.



If	Then
c_1: `hasElement` n_1 $\Diamond A$	a_1: `createNewNode` n_1 n_2
	a_2: `link` n_1 n_2 R
	a_3: `add` n_2 A

Fig. 1. LoTREC declarative language for rewriting rules vs usual graphical notation

The objects of S_{L_ρ} are related by a set of constraint predicates, called *elementary conditions*. A set of *elementary actions* describes both the changes that should be made on these objects, and the new objects which should be created and added to L_ρ, in order to obtain the *replacement* graph R_ρ. Formally:

Definition 3 (Elementary Conditions). An elementary condition c of a given rule ρ is a first order predicate of the form $\texttt{keyword}(p_1, \ldots, p_n)$ where $\texttt{keyword} \in \mathcal{K}^n$ and $p_i \in Param(c)$ s.t. $Param(c) \subseteq S_{L_\rho}$, for $i \in \{1, \ldots, n\}$.

An action a is defined in the same way. We omit the parentheses and commas from these definitions for the sake of simplicity. The complete list of LoTREC predefined conditions and actions is given in [5].

Example 1. The condition c_1 given in figure 1, `hasElement` n_1 $\Diamond A$, is a well defined condition in LoTREC. It takes in two parameters: a node (n_1) and a formula ($\Diamond A$). Intuitively, such a condition is used to ensure that, in a matching subgraph g, the condition $\Diamond A \in f_g(n_1)$ is satisfied.

Definition 4 (Rewriting rule). A rewriting rule ρ is a pair of ordered sets: $LHS_\rho = (c_1, \ldots, c_l)$ of *elementary conditions*, and $RHS_\rho = (a_1, \ldots, a_r)$ of *elementary actions*.

Note that $|LHS_\rho|$ is equivalent to $|L_\rho|$, since n conditions at most are sufficient to instantiate n different variables in LoTREC.

In order to allow the verification of a condition c on a given graph g, a subset of $Param(c)$ should be already instantiated by elements of g. We denote this set by $Activation(c)$. During this verification, the remaining parameters will also be instantiated. The set of these parameters is called $Update(c)$. It is clear that $Param(c) = Activation(c) \cup Update(c)$, and $Activation(c) \cap Update(c) = \emptyset$.

Example 2. Considering the condition c_1, of figure 1, the instantiation of $\Diamond A$ should be preceded by the assignment of a concrete instance node to n_1. Thus it is obvious that $Activate(c_1) = \{n_1\}$ and $Update(c_1) = \{\Diamond A\}$.

Definition 5 (Local search plan). A local search plan for a rule ρ is an ordered set $p = (c_1, \ldots, c_n)$ defined over the set of conditions LHS_ρ, such that: $n = |LHS_\rho|$, i.e. all the conditions of ρ are included in p; and $\forall\ i \in \{2, \ldots, n\}$, $Activate(c_i) \subseteq \cup_{k=2}^{i-1}\ Update(c_k)$, i.e. that the needed parameters for a given condition c_i are assured by the former conditions c_k s.t. $k < i$.

We call c_1 the head of p, while $(c_2, \ldots, c_n)$ is called the tail of p (see figure 2). Two plans are equivalent if they have the same head condition, regardless the order of the tail conditions. They are called distinct otherwise.

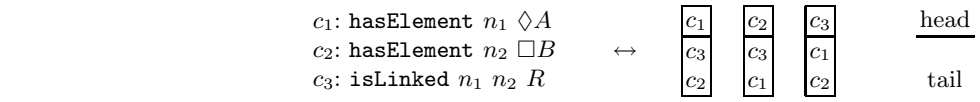


Fig. 2. Rules compilation: ordered lists of conditions are the local search plans

Rules Compilation The compilation of a rule ρ is the computation of the set of all possible distinct local search plans of ρ. This set is constructed in polynomial time before starting the rewriting process, since a rule ρ has at most $|LHS_\rho|$ plans, i.e. $|L_\rho|$ plans.

Invariant, Added and Removed Symbols We use in the sequel $Inv_\rho = S_{L_\rho} \cap S_{R_\rho}$ to denote the set of *invariant* symbols in ρ, i.e. the objects that are preserved during ρ application; $Add_\rho = S_{R_\rho} \setminus Inv_\rho$ to denote the set of newly created symbols i.e. the objects that will be *added* to the host graph; and $Rem_\rho = L_\rho \setminus Inv_\rho$ to denote the set of existing objects that will be *removed*. In general $R_\rho = (L_\rho \setminus Rem_\rho) \cup Add_\rho$, whereas in LoTREC, $R_\rho = L_\rho \cup Add_\rho$ since there are no *remove* actions, i.e. for every rule ρ, $Inv_\rho = L_\rho$.

Rule application A *transformation step* of a rewriting rule is accomplished, as shown in figure 3, by first matching L_ρ against a subgraph of the current working graph called the *redex*, and then replacing it by a copy of R_ρ.

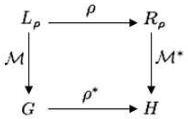


Fig. 3. Rewriting rule application in Single Push-Out approach

Definition 6 (Redex). The redex of L_ρ w.r.t. $\mathcal{M}$ is $g \subseteq G$ s.t. $\mathcal{M}(L_\rho) = g$.

Property 1. A redex g verifies the conditions of LHS_ρ w.r.t. the variable instantiation $\mathcal{M} : S_{L_\rho} \rightarrow S_G$, namely $g \models_{\mathcal{M}} LHS_\rho$.

Definition 7 (Replacement). The replacement g' of a redex g, $g' = \rho^*(g) = \mathcal{M}^*(R_\rho)$, is computed w.r.t $\mathcal{M}^* : S_{R_\rho} \rightarrow S_H$; namely $g' \models_{\mathcal{M}^*} RHS_\rho$, such that: $\mathcal{M}^*(s) = \mathcal{M}(s)$, for every $s \in S_{L_\rho}$, otherwise $\mathcal{M}^*(s) = C_s$ where C_s is a new (constant) object introduced to S_H.

It our case $g \subseteq g'$, and H can be viewed as G doted by new objects created and added by ρ, i.e. $H = G \cup \mathcal{M}^*(Add_\rho)$. However, we can extend this definition, in order to deal with *remove actions*, to : $H = (G \setminus \mathcal{M}^*(Rem_\rho)) \cup \mathcal{M}^*(Add_\rho)$ by using partial morphisms instead of total ones.

Example 3. Consider the subgraph $g = (\{N_1\}, \emptyset, \{\Diamond\Box q\}, s, t, f)$ where $f(N_1) = \{\Diamond\Box q\}$. It consists of a single node N_1 containing the formula $\Diamond\Box q$. The application of the rule of figure 1 on g leads to the graph $g' = (\{N_1, N_2\}, \{E\}, \{\Diamond\Box q, \Box q, R\}, s', t', f')$ where N_2 is a new created node, E is a new created edge, $s'(E) = N_1$, $t'(E) = N_2$, $f'(N_1) = f(N_1)$, $f'(N_2) = \{\Box q\}$, $f'(E) = \{R\}$.

Strategies In LoTREC, we make no choice among matching subgraphs, so that a transformation step consists of applying a given rule wherever it is possible in the host graph. However, the choice of the rule is made by a user-defined strategy. A strategy can call a set of rules, other strategies or a set of simple control structures, called *routines*, such as `Repeat`, `All-Rules` and `First-Rule` (for more details see [2]).

2 Event-driven Pattern Matching Process

Events When an elementary action is applied on a subgraph of the host graph, it launches an *event* of a corresponding type which embeds information about the action parameters. Thus the events are defined similarly to conditions and actions (c.f. definition 3). The graph objects embedded in an event can be viewed as *pins* used to fix the value of one or more variables of the LHS. In this way, the other variables are instantiated by searching, locally around these pins, for their convenient matches.

Example 4. The application of a_1, a_2 and a_3, of the example 3, generate respectively the events $E_1 = \texttt{NodeCreated}(N_1, N_2)$, $E_2 = \texttt{NodesLinked}(N_1, N_2, E)$ and $E_3 = \texttt{FormulaAdded}(N_2, \Box q)$.

Events Dispatching Once launched due to a given rule application, an event is dispatched at run-time to every rewriting rule, including the rule that launches it. Each rule has its own *events queue*, which is simply an ordered set of events. The events received by a given rule are enqueued at the end of its events queue.

Local Redex Initialisation and Completion When it is called by the strategy, a rule treats the events, stored in its queue since the last strategy call, with respect to their arrival order. Given an event, exploring a local search plan consists of processing the event with the head condition to establish an partial redex. Partial redexes are completed during the verification of the tail conditions.

Example 5 (Redex initialization). Processing the event E_3 of the example 4 with the head condition c_2:`hasElement` n_2 $\Box B$ of the middle search plan of figure 2,

succeeds in initializing the partial redex ($n_2 = N_2$ and $B = q$) in the graph g' of example 3. Verifying C_3 on this redex is possible with ($n_1 = N_1$). Verifying C_1 on this last partial redex also succeeds, and completes it with ($A = \Box q$).

Discussion and Conclusion

A main difference between this system and general purpose systems is that the formula instantiation is integrated within the pattern matching process. This makes the experimental benchmarks difficult to design, and thus only theoretical results are discussed.

Comparing to PROGRES [8], the compilation of a rule ρ leads to $|L_\rho|!$ different plans, while in LoTREC it is bound by the size of its left-hand side $|L_\rho|$. In addition, although PROGRES supports an incremental technique called attributes update, this technique detects only the invalidation of possible variable assignments. Thus it does not exploit the whole information embedded in the changes made on the graphs. In LoTREC we use these information to detect new possible valid assignments.

Similarly to incremental update [7], our method avoids restarting already-done pattern matching processes. However, our technique is less space and time-consuming. In fact, in our system, a rule keeps only the events occurred since its last application, and releases them all after being applied once again. Whereas the incremental update needs to keep successful matches in tables during the whole run-time. Furthermore, these tables need a considerable amount of pre-processing at initialization time and they are maintained by continuous updates, while the events stored temporary in our rules need neither initialization nor update, and thus have no additional time-cost.

In this paper we present a graph rewriting system adapted to theorem proving by tableau for modal logics. We propose a local and event-driven mechanism for limiting the effect of the graph pattern matching problem. At a given rewriting step, we establish the match process only on the local subgraphs changed by the rules during the previous step. It is clear that this technique has no advantage in specific applications where the redexes matched at each step are as numerous as possible. However, in the applications where the rules are applied alternatively in k redexes at each step where k is likely to be far less than N, our technique reduces the N factor of $O(|N^L|)$ time complexity of this problem to k.

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank David Fauthoux who established the conception of the LoTREC rewriting system.

References

1. M. A. Castilho, L. Farinas del Cerro, O. Gasquet, and A. Herzig. *Modal Tableaux with Propagation Rules and Structural Rules.* Fundamenta Informaticae, 1997

2. L. F. del Cerro, D. Fauthoux, O. Gasquet, A. Herzig, D. Longin, and F. Massacci. *Lotrec: The Generic Tableau Prover for Modal and Description Logics*. Lecture Notes In Computer Science, vol. 2083. Springer-Verlag, 2001
3. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. *Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java*. Lecture Notes In Computer Science. Springer-Verlag, 1998
4. M. Rudolf. *Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching*. Lecture Notes In Computer Science, Springer-Verlag, 2000
5. M. Sahade. *The conditions and actions in LoTREC language*. Tech. report, 2004
6. G. Taentzer. *Adding visual rules to object-oriented modeling techniques*. In Proceedings of Technology of Object-Oriented Languages and Systems, 1999.
7. G. Varró and D. Varró. *Graph Transformation with Incremental Updates*. Electronic Notes in Theoretical Computer Science, 2004
8. A. Zündorf. *Graph Pattern Matching in* PROGRES. Lecture Notes In Computer Science. Springer-Verlag, 1996

Editing Nested Constraints and Application Conditions

Karl Azab

Carl v. Ossietzky Universität Oldenburg, Germany *
azab@informatik.uni-oldenburg.de

Abstract. Nested constraints and application conditions express first-order properties on graphs and are implemented in the system ENFORCe. While ENFORCe has been without a GUI, ROOTS is a new GUI for the graph transformation engine of AGG and provides standard editing functionality for graphs and graph transformation rules. We integrated ENFORCe with ROOTS to get these editing features and implemented a tree-oriented visualization for editing nested constrains and application conditions.

1 Introduction

Graph programs use graph transformations to compute relations on graphs [1]. Graph transformations is a formalism which describes how graphs are rewritten by rules with graphs as left and right hand sides. An overview of the computations that can be made by graph transformations is given in [2–4].

We use nested constraints and applications conditions – (nested) graph conditions for short – as described in e.g. [5] as a visual representation of first-order logic formulas on graphs. As constraints, graph conditions express properties on graphs and as application conditions they limit the applicability of graph transformation rules. Given a (double-pushout) rule, there are transformations of constraints into application conditions that limit the applicability of that rule in such a way that the constraint is satisfied if the rule can be applied [6]. When graph conditions specify pre- and postconditions on graph programs, a weakest precondition can be computed [7].

ENFORCe [8] (ENsuring FORmal Correctness of high-level programs) implements the transformations and static analysis methods for nested graph conditions mentioned above. Transformations are done on a categorical level with the help from structure specific plug-ins (also called *engines*). ENFORCe has an engine for directed labeled graphs but input and output from ENFORCe has been

*This work is supported by the German Research Foundation (DFG) under grant no. HA 2936/2 (Development of Correct Graph Transformation Systems). The author thanks Stefan Jurack for making the ROOTS software available and providing technical assistance. We also thank Annegret Habel and Karl-Heinz Pennemann for commenting on a draft of this paper.

via text files which need to be manually created and interpreted. This is of course a tedious and error prone method. To more conveniently access ENFORCe, a GUI for graphs, rules, and nested graph conditions is needed.

ROOTS [9] (Rule based Object Oriented Transformation System) is a new GUI for the graph transformation engine of AGG [10]. It contains editors for e.g. graphs and rules and supports graph layout. ROOTS is based on an extensible EMF model which is only loosely coupled to AGG and can therefore relatively easily be ported to other graph transformation tools. While ROOTS have editors for atomic constraints and negative application conditions, it does not natively support nested constraints and application conditions.

There are other GUIs for graph transformation systems, e.g. Fujaba [11], with established plugin architectures, but both ROOTS/AGG and ENFORCe work on double-pushout graph transformations and their verification techniques. In a longer perspective, we hope this similarity will have a synergetic effect on the integration's usability.

This paper describes the integration of ENFORCe and ROOTS and the implemented extensions necessary for editing nested graph conditions. The integration was done at the interface between ROOTS and AGG: synchronization routines were added for the features in ROOTS that are also available in ENFORCe. The new editor extensions for graph conditions are synchronized to ENFORCe's internal data structures.

The paper is organized as follows: In Sect. 2 we review graph conditions and explain the visualization approach we used for nested graph conditions. Section 3 describes how ENFORCe was integrated with ROOTS, the implemented extensions to ROOTS and show how existing view components could be reused with only small changes. We conclude our work in Sect. 4.

2 Layout of Graph Conditions

In this section we first review the formal definition of graph conditions and then show the basic idea of how the implemented editor visualizes them. We define graph conditions as in [5–7] but omit the standard definitions of graphs and graph morphisms, and point the reader to e.g. [12] for details.

Definition 1 ((nested) graph conditions). A *graph condition* over a graph P is of the form $\exists a$ or $\exists(a, c)$, where $a\colon P \to C$ is an injective graph morphism and c is a condition over the graph C. Moreover, Boolean formulas over conditions (over P) are conditions (over P). A morphism $p\colon P \to G$ *satisfies* a condition $\exists a$ ($\exists(a, c)$) over P if there exists an injective morphism $q\colon C \to G$ with $q \circ a = p$ (satisfying c). A graph G *satisfies* a condition $\exists a$ ($\exists(a, c)$) if all injective morphisms $p\colon P \to G$ satisfy the condition. The satisfaction of conditions over P by graphs or morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We use an abbreviation $\forall(a, c)$ to denote $\neg\exists(a, \neg c)$.

Since we are interested in editing, we will focus on the syntactic structure of graph conditions, the reader interested in the semantics of nested graph conditions is referred to e.g. [5]. We use graph conditions in different contexts – on graphs, then called constraints and graph transformation rules, then referred to as application conditions, see e.g. [5, 6] for details.

To demonstrate, let us model two responsibilities of an operating system: scheduling and resource allocation. A core, process, and resource is represented by a node labeled C, P, and R, respectively. That a core is devoted to executing the threads of a particular process is modeled by an edge from a C-node to a P-node. A request from a process to obtain a resource is modeled by an edge from a P-node to an R-node. That a resource has been assigned to a process is modeled by an edge from an R-node to a P-node. We can formulate two desirable properties for this model: (1) *A resource may at most be assigned to one process*, which expressed as a graph condition looks like $\neg\exists(\emptyset \to$ and (2) *Every core is assigned to run threads in exactly one process*, shown in Fig. 1.

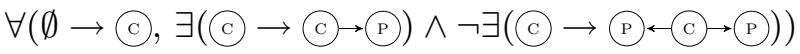


Fig. 1. A nested graph condition for scheduling.

As already becoming apparent from the condition in Fig. 1, large graph conditions can be difficult to read. To present a survey of a graph condition, we represent graph conditions as trees: A quantifier together with its morphism becomes a node and its nested condition becomes its child. Boolean formulas are represented as trees in the obvious way. To simplify the layout and reuse existing view components, we show the graph condition tree in two views: one for the tree structure and one for morphisms. We then only need to layout trees and morphisms separately – and tree layout is easy and for morphisms the only difficult part is to layout the domain and codomain, and functionality for graph layout already exists in ROOTS. The tree layout for the nested graph condition from Fig. 1 is shown in Fig. 2 (a). The separated tree and morphism view of that graph condition is shown in Fig. 2 (b). In the latter figure, a, b_1 and b_2 identifies the entry point of the morphism view into the tree view.

3 Implementation

ROOTS is implemented as an Eclipse plugin and is based on the model-view-controller pattern, where the model is generated by EMF [13]. That generated model is supplemented with routines to synchronize it with AGG's internal data structures. Our integration adds synchronization routines to ENFORCe for the common functionality of AGG and ENFORCe. The extensions for graph conditions are only synchronized to ENFORCe, as there is no suitable data structure

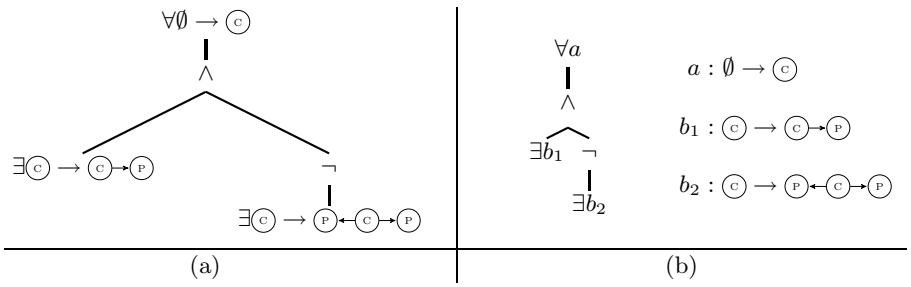


Fig. 2. (a) Tree representation and (b) two views of of a condition.

available in AGG. In this way, many editing features become (for the end-user) seamlessly available also for ENFORCe, see Fig. 3. An advantage of using an ap-

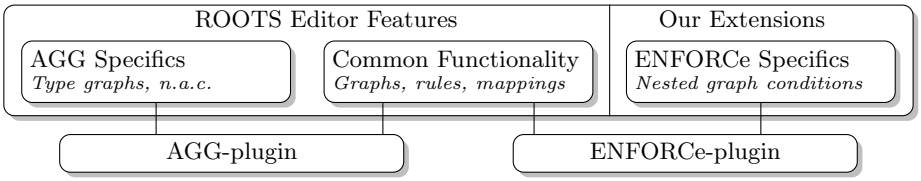


Fig. 3. Model synchronization.

plication framework like EMF is that some features come for free: Persistence of model states (i.e. saving and loading). Furthermore, models generated by EMF implements the Observer pattern, meaning that changes made in the model by e.g. a result from ENFORCe will automatically be propagated to its observers (yielding the backwards synchronization from ENFORCe to the views). EMF also makes it easy to extend ROOTS' underlying EMF model to work with graph conditions.

ROOTS and AGG are software for typed attributed graphs while ENFORCe works on directed labeled graphs. For a smooth port, we disabled ROOTS' model editor and used a static model consisting of a singular symbol type and link type, each with one attribute: a string for labels.

For graph conditions, we extended ROOTS' EMF model by a few interfaces, the inheritance hierarchy for those are shown in Fig. 4. The two interfaces `ROOTS` and `Rule` are part of the original ROOTS model (see [14] for details), the others are our extensions. `ROOTS` collects graphs, rules, etc, into a graph transformation system, and `GCROOTS` adds nested graph conditions and rules with left and right application conditions to these transformation systems.

The already existing interface for negative application conditions were not expanded upon, since its view in ROOTS is too different from the morphism view we are aiming for – our morphism view is more similar to ROOTS' rules. `GCMorphism` is a generalization of the nodes with morphisms in the tree view and therefore extends `Rule`, this allows us to reuse the components for editing a rule's left and right hand side to the editing of the morphism's domain and codomain. `NestedGraphCondition` specifies the nesting of graph conditions and is therefore a supertype of all types that are represented as nodes in the tree view. The original ROOTS model already supports binary Boolean formulas, but our conditions are best represented by junctions on sets of conditions. Boolean formulas are therefore represented by the new types `GCConjunction`, `GCDisjunction`, and `GCNegation`.

`GCRule` extends normal rules with left and right application conditions, `GCLAC` and `GCRAC` – both consisting of a set of `NestedGraphCondition` objects. The interface `GCContainer` specifies the possible contexts of where a `NestedGraphCondition` may occur.

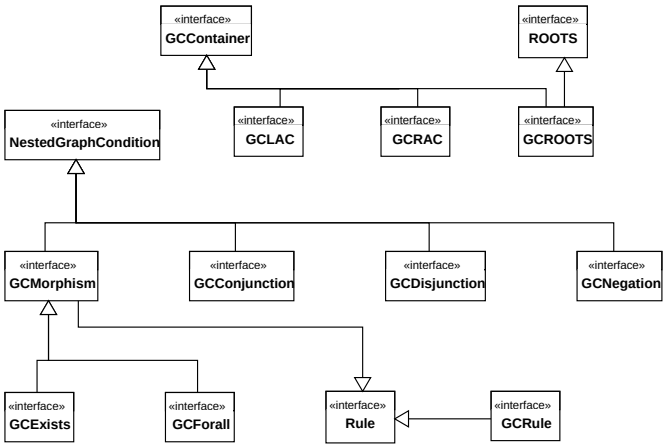


Fig. 4. Model extension for editing graph conditions.

Recalling the two views discussed in Sect. 2, we now discuss a few issues regarding their implementation in ROOTS. The tree view was implemented as an extension to ROOTS' existing tree editor for graphs, rules, etc. As an example, we show the tree view from Fig. 2 (a) in the bottom of the left area in Fig. 5, named GraphCondition there. The right area of Fig. 5 shows the morphism view – the reused rule view – of the morphism identified by b_2 in Fig. 2 (b).

Since graph conditions can also be used in a rule context, the corresponding visual additions where made to a special kind of rule with left and right application condition. Figure 5 compares the previous rules with negative application

conditions (named Rule in that figure) and the new rules with left and right application conditions (there named Rule w. a.c.).

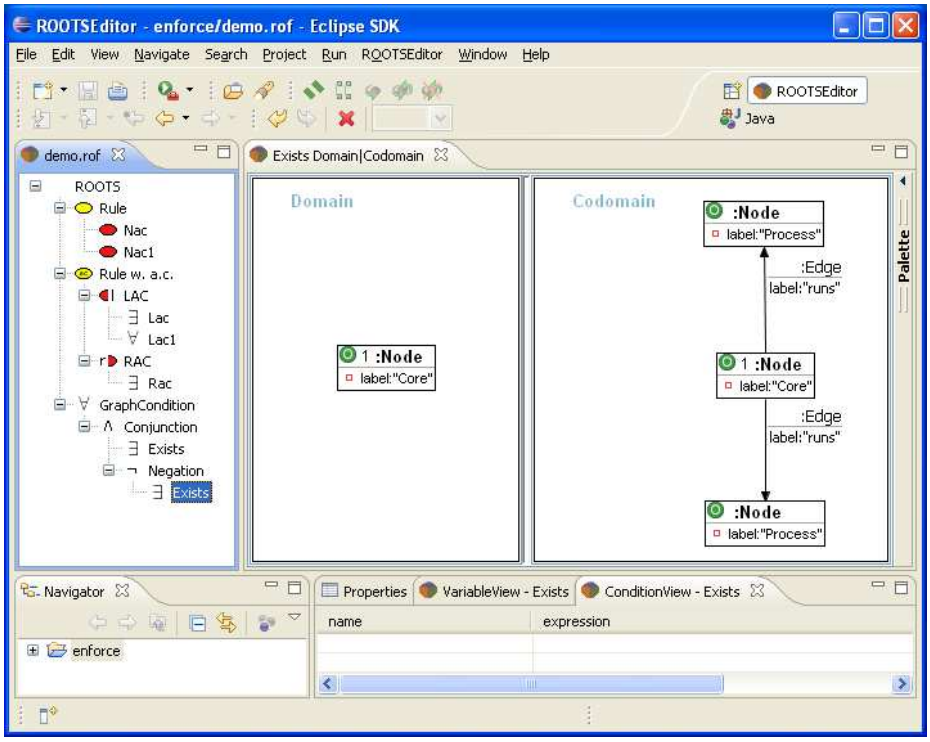


Fig. 5. Editing a nested constraint.

The morphism view is implemented by making some extensions to the rule editor of ROOTS, e.g. when a nested condition is created, it is by default set to the identity morphism of the codomain of its closest ancestor of a `GCMorphism` subtype. Commands for editing the trees and opening the morphism editor from the tree view are integrated with ROOTS' standard mouse operations.

4 Summary

We reported on an integration of ENFORCe and the ROOTS GUI for graph transformation systems and described an extension for graph conditions. Our layout approach were based on the nesting structure of graph conditions and was separated into two views, one for nesting and one for morphisms. We could therefore reuse existing ROOTS components and visually integrate the extension

with other GUI elements. The resulting editor is however still quite similar to the representation used in e.g. [7].

It would be interesting to have a plug-in architecture in ROOTS that allows new features to be integrated more easily. Plug-ins would initially report to ROOTS on the particular category they support. ROOTS could then better layout objects of the particular category and report on what tool features are available to which categories. This would also allow the editor to be generalized to work on nested high-level conditions. Since the tree structure and mapping mechanism can work on objects from any HLR category, only the object visualization needs to be updated for new categories.

References

1. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001). Volume 2030 of Lecture Notes in Computer Science., Springer-Verlag (2001) 230–245
2. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)
3. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
4. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 3: Concurrency, Parallelism and Distribution. World Scientific (1999)
5. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science (2008) To appear.
6. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae **74(1)** (2006) 135–166
7. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: Graph Transformations (ICGT 2006). Volume 4178 of Lecture Notes in Computer Science., Springer-Verlag (2006) 445–460
8. Azab, K., Habel, A., Pennemann, K.H., Zuckschwerdt, C.: ENFORCe: A system for ensuring formal correctness of high-level programs. In: Proc. of the Third International Workshop on Graph Based Tools (GraBaTs'06). Volume 1 of Electronic Communications of the EASST. (2006)
9. Jurack, S., Taentzer, G.: ROOTS: An Eclipse Plug-in for Graph Transformation Systems based on AGG. In: Pre-proc. of the Third International Symposium, Applications of Graph Transformations with Industrial Relevance (Agtive 2007). (2007)
10. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: Language and environment. In: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2. World Scientific (1999) 551–603
11. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool integration at the meta-model level within the fujaba tool suite. International Journal on Software Tools for Technology Transfer (STTT) **6** (August 2004) 203–218

12. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Graph-Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Springer-Verlag (1979) 1–69
13. Foundation, T.E.: Eclipse application frameworks. Web pages available at `http://www.eclipse.org/{emf,gef}` Visited June 3, 2008.
14. Jurack, S.: Konzeption und Implementirung einer Entwicklungsumgebung für regelbasierte Graphtransformationssysteme basierend auf AGG und Eclipse (2007) Master thesis, TU Berlin.

Parallel Graph Transformations in Distributed Adaptive Design

Leszek Kotulski[1] and Barbara Strug[2]

[1] Department of Automatics, AGH University of Science and Technology
Al. Mickiewicza 30, 30 059 Krakow, Poland
[2]Department of Physics,Astronomy and Applied Computer Science,
Jagiellonian University, Reymonta 4, Krakow , Poland,
kotulski@agh.edu.pl, barbara.strug@uj.edu.pl

Abstract. In this paper a graph transformation system with the parallel derivation is used to model the process of distribution and adaptation for computer aided design. It is based on earlier research in formal language theory, especially graph grammars, and distributed models. The motivation for the ideas presented here is given and possible ways of application are described. The application of this idea by the GRADIS multi-agent framework is also considered**.** The theoretical approach is illustrated by the example from the domain of flat layout design**.**

1. Introduction

This paper deals with a linguistic approach [2] to distributed parallel design. The formal model of computer-aided design is based on graph structures for which a lot of research has been done in context of design [3]. Semantic knowledge about the design is expressed by attributes. Graphs representing design object structures are generated by grammars which consist of graph rules.

Usually a complex design problem is divided into a number of subproblems. Therefore we propose a distributed design system consisting of several cooperating graph grammars. Searching for each subproblem solution is supported by one grammar. Solving the whole problem requires the ability of grammars assigned to subproblems to communicate [4,5,6,7,8,9].

Our first attempt to solve the problem of cooperation strategy, where the sequence (in which component grammars are activated) is determined by a control diagram, is presented in [10,11,12]. In this paper the system communication is realized by a specially introduced agents the notion of complementary graphs [13] and conjugated graph grammars [14]. The proposed approach is illustrated on the example of designing house layouts and internal rooms arrangements

2. GRADIS agent model.

The GRADIS framework (that is an acronym of GRAph DIStribution toolkit) makes possible the distribution of a centralized graph and the controlling its behavior with the help of concurrent processes. The proposed solution is based on multiagent approach; an agent is responsible for both a modification of the maintained (local) graph, in a way

described by the graph transformation rules associated with it and a cooperation with other agents in order to maintain the cohesion of the graph system. The GRADIS agent model assumes the existence of two types of agents, called: maintainers and workers.

The maintainer agent – maintains the local graph; the whole set of maintainers takes care of the system cohesion understood as a behavior equivalent to the graph transformations carried out over the centralized graph. Initially we assume, that at the beginning one maintainer controls the centralized graph, but it is able to split itself into a set of maintainer agents controlling parts of the previous graph. The cooperation of the maintainers is based on the exchange of information among the elements of the agent's local graph structure; the graph transformation rules are inherited from the centralized solution.

The worker agents are created: temporarily, in order to carry out a given action (eg. to find a subpattern), or permanently - to achieve a more complex result (eg. for detail design of an element represented at the lower graph hierarchy). The worker's graph structure is generated during its creation (by a maintainer agent or another worker agent) and is associated with a part of the parent's graph structure. This association in not one between some nodes of a graph structure maintained by these agents. The parent-worker association is done at the graph transformation level i.e.some worker's transformations enforce the application of some graph transformations over the parent's graph structure.

2.1. Complementary graph as a background for maintainer agent work

The data structure, that is maintained and transformed by agents, has a form of labeled (attributed) graphs. Let Σ^v and Σ^e be a sets; the elements of Σ^v are used as node labels and the elements of Σ^e are used as edge labels. The graph structure is defined as follows:

Definition 2.1

A (Σ^v,Σ^e)-graph is a triple (V,D,v-lab) where V is a nonempty set, D is a subset of $V\times\Sigma^e\times V$, and v-lab is a function from V into Σ^v. ■

For any (Σ^v,Σ^e)-graph G, V is set of nodes, D is set of edges and v-lab is a node labeling function. It can be extended. by introduction attributing functions for both nodes and edges, but it has no influence on the graph distribution.

Our intention is to split the graph G into several parts and to distribute them to different locations. Transformations of each subgraph G_i will be controlled by some maintainer agent.

To maintain the compatibility of centralized graph with the set of split subgraphs some nodes (called *border nodes*) should be replicated and placed in the proper subgraph. We will mark a border node by a double circle. During the splitting of the graph we are interested in checking if the connection between two nodes crosses a border between subgraphs The algorithms for splitting a common graph and joining partial graphs into the common graph are presented in [13].

For any border node v in the graph G_i we can move boundary in such a way that, all nodes that are connected with v (inside other complementary graphs) are incorporated to G_i as a border nodes and the v node replicas are removed from another graphs (i.e. v

becomes a normal node). For graphs presented in Fig. 3.1b an *incorporate((0,1),1)* operation creates graphs presented in Fig. 3.1c.

The GRADIS framework associates with each distributed complementary graph the maintainer agent, that not only makes the local derivations possible, but also assures the coordination and synchronization of parallel derivations on different complementary graphs.

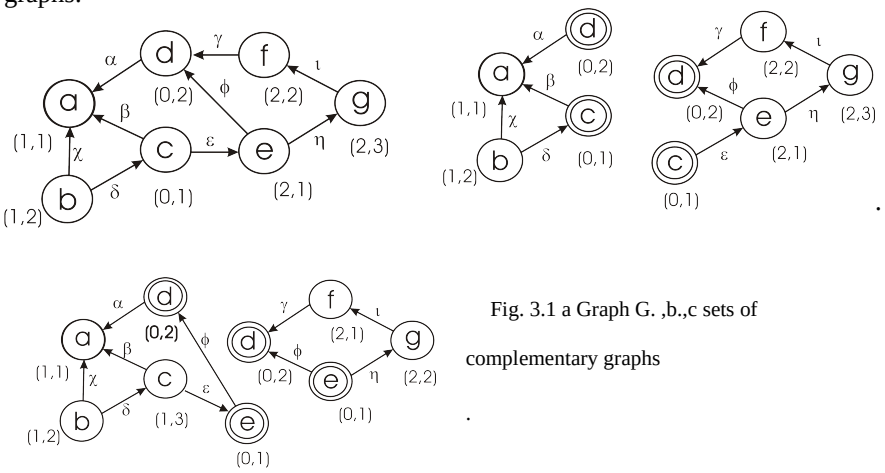


Fig. 3.1 a Graph G. ,b.,c sets of complementary graphs

.

The main idea is to either apply transformation locally or to create the appriopriate local environment (with the use of incorporate operation) in which to properly apply transformatons locally. The application of a transformation is strongly dependent on the graph transformation mechanism; for the ETPL(k) graph grammars the formal description is presented in [13] The presented approach does not depend on the specific properties of the graph transformation mechanism like NLC embedding transformation (in case of the algorithmic approach) [15] or single- and double-pushout (in the case of the algebraic approach) [16]. In [13,17] the cooperation among the agents in the case of these types of graph transformations is considered and a detail algorithms based on these transformations properties are presented.

2.2 Conjugated graph grammar as a background for workers agents cooperation.

The formal background for the worker agent cooperation is based on a conjugated graph grammars theory [14,18]. In the conjugated graph a new type of nodes appears – remote nodes (Fig. 3.2). There is fundamental difference between complementary and conjugate approaches: in the first one, the agents create a local environment for correct application of a given production (so there is no synchronization at the production level), in the second approach the local graphs that consist of partially replicated nodes are synchronized with the help of application of the conjugated transformation rules. More precisely remote nodes represent the nodes appearing in other graph structures.

In the conjugated graphs grammars we assume that, the graph transformations P (on the first agent's graph structure) in which a remote node *w* exists and is associated with the graph transformation Q (on the other agent's graph structure), such that it modifies the neighborhood of the node represented by w. The pair P and Q are called conjugated transformations in context of the remote node w. In order to synchronize the set of conjugated graph transformations we assume that, GRADIS assures that both P and Q graph transformations will be successfully performed. Here we assume that graph grammars are transactional conjugated graph grammars i.e. either both P and Q are applied or neither action is carried out.

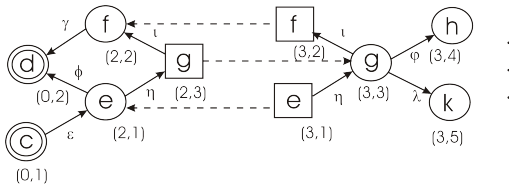


Figure 3.2 Associations in conjugated graphs.

Let's notice that in conditional and transactional models, there is no assumption on the type of graph transformation used by agents. There is no objection to constructing a system in which parent and son agents use different graph transformation systems.

3. Distributed Adaptive Design

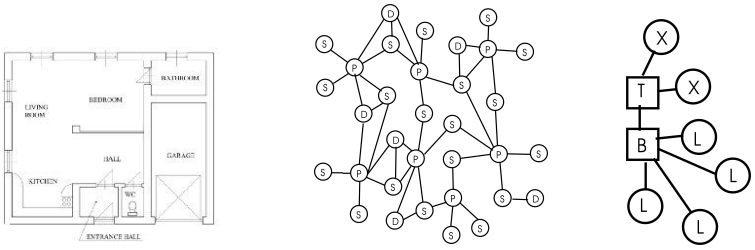
The model described in previous section is very appropriate for the design process. The example of such design process is presented below. For the clarity of presentation the design patterns are restricted to minimal elements that allows us to illustrate the introduced concept. In our example we use a flat layout and arrangement design problem. This process is composed of several phases: firstly we create basic plans (eg. a building), next we can plan rooms arrangement and finally details of the furniture. Graph transformations seem to be very promising in this application domain [10]. In fig 3.1a a layout of a flat is presented. A graph representing this layout at the highest level is presented in fig. 3.1b. This graph contains only nodes representing spaces (P), walls (S) and doors (D). It can be noticed that even for such small apartment and a very high level of representation this graph large. Adding to it nodes representing elements of furniture, and then parts of every single piece of furniture would make it rather overwhelming. So the use of approach from the GRADIS framework seems to be a good way to reduce amount of data that has to be taken into account at a time.

We assume that, the building plan is maintained by a set of maintainer agents, and each of maintainer agents takes care of the subgraph responsible for parameterization (allocation of walls, doors, windows, etc.) of one or more rooms. In case of design problems, the splitting is performed across "natural" borders. In house design problem nodes representing walls and doors seem to be good candidates for such natural border nodes. In fig. 3.2 three subgraphs resulting from splitting the full graph are depicted. It means that any operation including these nodes can not be carried out locally but must

involve cooperation with other agents maintaining subgraphs containing the same border nodes. Let's imagine that we want to move doors (change attributes associated with a node labeled by D). As D is a border node for any rule to be applied to it we must first carry out a sequence of incorporate operations. Each incorporate operation collects the neighbourhood nodes of D in other subgraph and then "moves" it into the graph in which the rule is to be applied. This operation thus moves the border and, at the same time, makes a node "D" internal one. Then the required transformation are applied. Moreover with each of the rooms it is associated a permanent worker agent that is responsible for this room arrangement. Let's note that the maintainer-worker relationship is more complex. Moving walls or doors has a strong influence on the room arrangement process and on the other hand putting the chair or a table again the wall with door is possible only when these door will be transferred to another place. Thus the cooperation between graph trans-formation systems supported by maintainer and worker agents should be very close.

In fig. 3.2 the subgraph representing one of rooms is depicted with a bit more details. From the point of view of the maintainer agent for this room only two nodes, B and T, are important as this agent is responsible for positioning the chair. This structure is maintained by agent responsible for designing a chair. An example of the structure for such an agent is depicted in fig 3.1c. Any change in this structure that does not influence the remote nodes is local

In fig. 3.2b a general hierarchy of workers and maintainers in a considered example is



presented

.

Figure. 3.1 a. A layout of a flat , b. a graph representing the layout from a. c internal structure of a chair

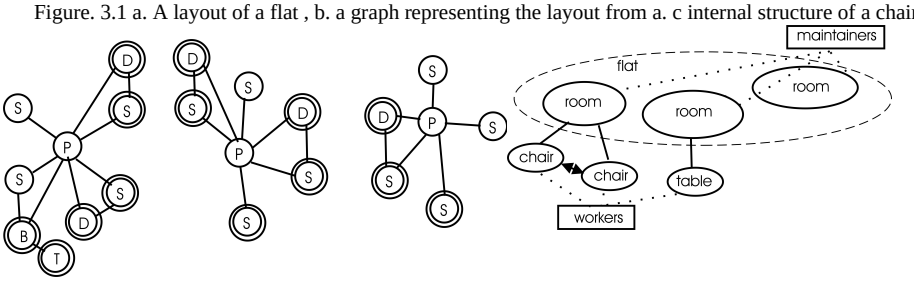


Figure 3.2 Three of 7 complementary graphs generated after splitting a full graph (fig. 3.1) and a diagram representing a hierarchy of maintainer and worker agents.

5. Conclusions.

In this paper a general framework for distributing large graph structures into smaller and thus easier to maintain and transform was presented It is worth noticing that this approach does not make any assumptions about the type of graph transformations that are used as rules at different levels (by different agents).

The proposed formalism was applied to the domain of distributed design that is in particular need of a good formalism that would allow for both distribution and adaptation within one general framework. The dynamic character of design in which external and internal constraints may require adaptation can be described by such a formalism. Moreover the problem of the subsystems modification is solved.

This paper lays a ground for further research in both general methodology and its application to the domain of dynamic design. The use of this method in design context both simplifies it and adds complexity. On one side by introducing natural borders for splitting and natural hierarchy of tasks for maintainers and workers it makes the problem simpler than in a more general case. On the other hand the domain of design adds its own problems, especially a potential for deep propagation of cooperation and lead to a cascade of requests.

In future we plan to test more intensively the different types of cooperation (conditional, transactional) and their influence on the working of this methodology in real situations.

6. Reference.

1. Rozenberg, G.: Handbook of Graph Grammars and Computing By Graph Transformation: Volume I, Foundations. Ed. World Scientific Publishing Co., NJ, (1997)
2. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G.: Handbook of Graph Grammars and Computing By Graph Transformation: Volume II, Application, Languages and Tools. Ed. World Scientific Publishing Co., NJ, (1999)
3. E.Grabska, W. Palacz. Hierarchical graphs in creative design. MG&V, 9(1/2), pp. 115-123, (2000)
4. E. Csuhaj-Varju and Gy. Vaszil. On context-free parallel communicating grammar systems: Synchronization, communication, and normal forms. Theoretical Computer Science, 255(1-2), pp. 511-538 (2001).
5. E. Csuhaj-Varju, J. Dassow, J. Kelemen and Gh. Paun. Grammar systems. A grammatical approach to distribution and cooperation. Topics in Computer Mathematics 8. Gordon and Breach Science Publishers, Yverdon (1994)
6. E. Csuhaj-Varju: Grammar systems: A short survey, Proceedings of Grammar Systems Week 2004, 141-157, Budapest, Hungary, July 5-9 (2004).
7. J. Kelemen. Syntactical models of cooperating/distributed problem solving. Journal of Experimental and Theoretical AI, 3(1), pp. 1-10 (1991)
8. C. Martin-Vide and V. Mitrana. Cooperation in contextual grammars. In A. Kelemenov, editor, Proceedings of the MFCS'98 Satellite Workshop on Grammar Systems, pp. 289-302. Silesian University, Opava (1998)
9. M. Simeoni and M. Staniszkis. Cooperating graph grammar systems. In Gh. Paun and A. Salomaa, editors, Grammatical models of multi-agent systems, pp. 193-217. Gordon and Breach, Amsterdam (1999)

10. E. Grabska, B. Strug. Applying Cooperating Distributed Graph Grammars in Computer Aided Design, Parallel Processing and Applied Mathematics (PPAM 2005), LNCS vol 3911, pp. 567-574 Springer (2006)
11. E. Grabska, B. Strug. G Slusarczyk, A Graph Grammar Based Model for Distributed Design, International Conference on Artificial Intelligence and Soft Computing, Artificial Intelligence and Soft Computing, EXIT, Warszawa 2006, pp 440-445 (2006)
12. Kotulski L., Strug B.: Distributed Adaptive Design with Hierarchical Autonomous Graph Transformation Systems. ICCS 2007, LNCS 4488, pp. 880-887, Beijing(China) (2007)
13. Kotulski L.: GRADIS – Multiagent Environment Supporting Distributed Graph Transformations. M. Bubak et all (eds.): ICCS 2008, Part I, LNCS 5101, pp. 386-395, 2008.
14. Kotulski L., Fryz Ł: Conjugated Graph Grammars as a Mean to Assure Consistency of the System of Conjugated Graphs. Accepted at the Third International Conference on Dependability of Computer Systems DepCoS – RELCOMEX (2008)
15. Engelfriet, J., Rozenberg, G.: Node Replacement Graph Grammars, 3-94 in [1]
16. Ehrig H., Heckel R. Löwe M., Ribeiro L., Wagner A.: Algebraic Approaches to Graph Transformation – Part II: Single Pushout and Comparison with Double Pushout Approach. In [1] pp. 247-312.
17. Kotulski L.: Distributed Graphs Transformed by Multiagent System. L. Rutkowski et all(Eds.): ICAISC 2008, LNAI 5097, pp. 1234-1242, 2008.
18. Kotulski L., Fryz Ł: Assurance of system cohesion during independent creation of UML Diagrams. Proc. 2nd International Conference on Dependability of Computer Systems DepCoS - RELCOMEX 2007, pp. 51-58, IEEE Computer Society P2850, , Poland (2007)
19. Kotulski L., Sędziwy A.: Agent framework for decomposing a graph into the subgraphs of the same size, accepted to WORLDCOMP'08 (FCS'08) Las Vegas, USA (2008).

Visidia: An Environment for Programming Distributed Algorithms

Mohamed Mosbah

University of Bordeaux 1

Abstract. Visidia is a software tool that has been developed in order to help in the design, the experimentation, the validation and the visualization of distributed algorithms described by relabelling systems. Visidia allows the user to model a network, to implement and to execute a relabelling system.
A friendly Graphical User Interface allows the user to draw by "drag and drop" a graph which model the network. The user can add, delete, or select vertices, edges or subgraphs. Visual attributes of vertices and edges such as labels, colors or shapes have default values, but they can be easily customized. The tool provides a library of high level primitives to program the corresponding local computations .These primitives include the synchronizations between vertices depending on the type of local computation, the communication between processors such as send and receive messages, etc. The user can write an algorithm and execute it by pressing on appropriate buttons provided by the interface. The system automatically creates and assigns to each vertex a java thread which will run a copy (a clone) of the code implementing the relabelling system. The user can observe the messages exchanged between vertices (threads), and their states. In particular, label changes of vertices can be seen on-the-fly. The whole algorithm is animated in such a way that the user can follow its execution. Moreover, the number of exchanged messages is computed and displayed. This can be used to perform experiments for particular distributed algorithms and obtain statistical parameters. A control panel allows the user to play animation, pause it at any point during its execution, and stop it. Many distributed algorithms are already implemented and can be directly animated. Extensions to deal with agents or algorithms for sensor networks are under implementation. Visidia can be used as a platform for teaching or as a research platform for designing and validating distributed algorithms.
References:
[1] Bauderon, M., Metivier, Y., Mosbah, M., Sellami, A.: From local computations to asynchronous message passing systems. Research Report RP 1271-02, University of Bordeaux I (2002)
[2] Derbel, B., Mosbah, M., Gruner, S. Mobile agents for implementing local computations in graphs. In: Graph Transformations (ICGT 2008). LNCS, Springer (2008)
[3] http://www.labri.fr/visidia

AGG: A Tool Environment for Algebraic Graph Transformation

Gabriele Taentzer

University of Marburg

Abstract. The AGG tool environment consists of a graph transformation engine, several analysis tools for graph transformations and a graphical user interface for convenient user interaction. AGG supports the algebraic approach to typed, attributed graph transformation. It provides a typing concept for nodes and arcs which supports node type inheritance. Its attribution concept is based on Java expressions. Transformation rules may be equipped with negative application conditions. Rule applications may be controlled by graph constraints and explicit control constructs. Analysis tools offer graph parsing, critical pair analysis and applicability checks for transformation rules, as well as checking of termination criteria for controlled rule applications.

Author Index