# A Visual Modeling Technique for Controlling Graph Transformations*

S. GRUNER
*Technische Universität Berlin*

M. KURT
*Infonie GmbH Berlin*

G. TAENTZER
*Technische Universität Berlin*

**Abstract**

Sophisticated control concepts are necessary to make the application of graph grammars feasible for practical graph grammar engineering and specification tasks. For this purpose we introduce hybride control graphs based on the well known Activity Diagrams. Moreover, we present an interactive control graph tool showing the practical dimension of our concepts.

**Keywords**

graph grammars, non-determinism, control graphs, tool

## 1 Introduction

*Graph grammars* and graph transformation systems [9] have turned out to be applicable for solving various problems of recent computer science [3][4]. It is also known, however, that *pure* graph grammars are often *no* suitable solutions for practical problems, because

- graph grammar derivations (as any grammar derivations) are of non-deterministic nature whilst deterministic and predictable behavior is mostly required, and
- graph grammar derivations are local operations on some data graph whilst global operations are often required for reasons of consistency between different parts of the data graph which do not belong to the same neighborhood.[1]

*The project is partially supported by the ESPRIT Basic Research Working Group APPLIGRAPH and the German Research Council DFG.

[1]Given a graph replacement rule $r : L \rightarrow R$, a data graph $G$ and a match $m : L \rightarrow G$, the subgraph $m(L) \subseteq G$ is regarded as a neighborhood with respect to $r$.

For this reason, different concepts of *control* have been introduced in the graph grammar literature recently — in particular, we can mention the PROGRES system [13] and the GRRR system [11] which both are promising attempts to make the use of graph grammars feasible for various application tasks. Comparing the control techniques of PROGRES and GRRR we can observe that they are orthogonal to each other as shown in **Fig.1**:

- Whilst the *explicit* control concepts of PROGRES are powerful they support a *textual* notation only. This might be rather inconvenient from the user's point of view.

- The opposite is true in GRRR: there we can find *visual* control concepts the handling of which seems quite user-friendly, but their control behavior is built-in, thus: *implicit*, and cannot be modified by the user according to his requirements.[2]

| PROGRES | GRRR | New Approach |
|---|---|---|
| *explicit* | *implicit* | *explicit* |
| *textual* | *visual* | *visual* |

Figure 1: *comparison of control concepts*

Of course, there are many more concepts which cannot all be mentioned due to lack of space in this short paper, for example the *transformation units* of [8] or the *rule expressions* of [6]. The approach of [8] does not support visual control structures and is, thus, similar to PROGRES under this point of view. In the approach of [6], visual control structures are mentioned but not implemented by a software tool. The *story diagram* approach of FUJABA [5] is very related to our work and will be described below.

Starting from this discussion, we want to combine the advantages and avoid the disadvantages of both the control concepts of PROGRES and GRRR which means that we head for a controlling system of as well explicit and visual control structures. Based on the concept of *Activity Diagrams*, we present an almost completely implemented visual *control flow editor* for the attributed graph grammar system AGG [10] as the main contribution of this paper. The control structures specified with this editor can be interpreted and executed, and the user can watch the system running on the *control flow monitor* which is just another operational view of the control flow editor [7]. The conceptual basis of its *control flow diagrams* stem from a new combination of the widespread UML [14] and the historical Dijkstra schemas [2][1]. At the end of this introduction it is worth mentioning that the process of constructing control flow diagrams in the control flow editor

[2]Implicit control in GRRR means that the user can attach certain *flags* on certain nodes which invoke certain built-in graph replacement strategies. The replacement strategies themselves cannot be modified.



can be described by graph grammars again such that —back to the beginning— the methodological "circle" is closed.

## 2 Finding suitable Control Structures

Manipulating some data graph $G$ by a graph grammar $\mathcal{G} = \{G, r_1, \dots, r_n\}$, we have to face *two* different kinds of non-determinism, namely

- the choice of *what* rule $r_i$ shall be applied to $G$, and
- the choice *where* a selected rule $r_i$ shall be applied *within* $G$.

In the following, we are concerned with the first kind of non-determinism, but not with the second one. This means that we want to be able to determine a finite application sequence $r^{(1)}; r^{(2)}; \dots; r^{(m)}$ which we call a *graph transformation* on $G$ where $r^{(i)} \in \mathcal{G}$ for all $i = 1, \dots, m$. The second kind of non-determinism can be restricted by certain rule *parameters* determing partial matches already.

*Activity Diagrams* are a sub-language of the UML [14]. The well-known *program schemas* [1] —Dijkstra schemas are a special class of them— can be regarded as ancestors of that UML sub-language. In **Fig.2** a small example of an Activity Diagram is shown.

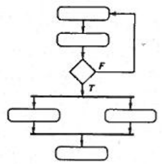


Figure 2: *sketch of an Activity Diagram*

As their original semantics is vague, Activity Diagrams can easily be interpreted in such a way that our control requirements are fulfilled. This has been suggested by the already discussed approach of [6] as well as by the approach of [5]. There, the action nodes of such "story diagrams" are attributed with whole graph transformation rules, whereas in our approach the action nodes are mainly attributed with rule *names* which makes our framework more scenario-independent and the diagrams less complex.

Please note that in the Activity Diagrams some non-determinism is re-introduced by the concept of arbitrary decisions (*fork*), which is not part of the Dijkstra terminology. *Well*-structured loops (*while*), however, are *not enforced* by the Activity Diagrams. Instead, the user is able to simulate not only *while* loops but arbitrary *goto* loops by (ab)using the concept of conditioned decisions (*if then else*) which Dijkstra has "considered harmful" for well-known reasons [2].

# 3 Control Flow Diagrams for Graph Transformation

In this section, we present our solution to the problem how graph grammar operations on data graphs can be controlled by visual means. Keeping Dijkstra's warning in mind, we *extend and restrict* the UML Activity Diagrams in such a way that their advantages are increased and their disadvantages are decreased. Then we explain how our version of *control flow diagrams* [3] are interpreted in a graph transformation environment.

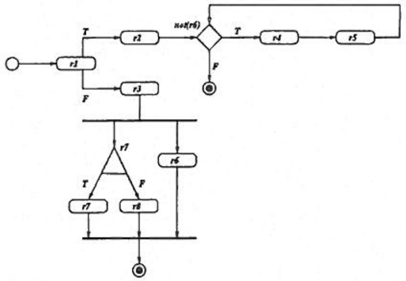


Figure 3: *example of a control flow diagram C for some graph transformation*

## 3.1 Structure

In our control flow diagrams, the *items* are connected by arrows $\rightarrow$ representing the control flow. The arrows may (but need not always) be attached with $T$ and $F$ symbols. There must be a *start* symbol $\circ$ and at least one *stop* symbol $\odot$. Furthermore we may have *rule application* symbols $\square$ and *if-then-else* symbols $\triangle$ (branching on pre-condition). Loops may be constructed with the *while* symbol $\diamond$ only, whilst the construction of loops with *goto* jumps via the $\triangle$ symbol is strictly forbidden in order to avoid badly structured spaghetti-diagrams. These symbols may be attached with logical expressions over some variables $r_i$. Finally we may *fork* and *join* the control flow (without any condition) by using the symbols $\top\top$ and $\bot\bot$. **Fig.3** shows an example of how all these symbols may be used; the definition can be found in [7].

## 3.2 Interpretation

The example of the control flow diagram $C$ given in Fig.3 is now explained in an informal way. Given a graph grammar $\mathcal{G} = \{G, r_1, \dots, r_8\}$ the rules $r_i$ of which

[3] The term "control flow diagram" is so widely used in computer science that we did not feel a need for inventing a special new name for our version of such diagrams.



shall operate on the data graph $G$. Given further an interpreter $I$ which is able to operate on $C$ and $\mathcal{G}$ by reading in $C$ and writing in $G$ according to the rules of $\mathcal{G}$.

After having started in $\circ$, the interpreter *tries* to apply $r_1$. If this trial *has been* successful (*post-condition* $T$), $I$ tries to apply $r_2$. Whether successful or not, $I$ enters the *while* structure $\diamond$. *As soon as* $r_6$ *is* applicable (*pre-condition* $F$), $I$ stops at $\odot$, otherwise the application of $r_4$ and $r_5$ is tried sequentially in the loop (pre-condition $T$), no matter if with success or without. If the trial of $r_1$ has not been successful, however (post-condition $F$), $I$ tries to apply $r_3$ and enters the fork $\top\top$ afterwards in any case. At this point, $I$ makes a non-deterministic choice by chance. *Either* rule $r_6$ is tried or the *if-then-else* structure $\triangle$ is entered: if $r_7$ is applicable (pre-condition $T$) this is done, otherwise (pre-condition $F$) the application of $r_8$ is tried. Finally, $I$ joins at $\bot\bot$ and stops at $\odot$.

In general, the decision symbols $\triangle$ and $\diamond$ may be attached with expressions in propositional logic ($\neg, \wedge, \vee$) over the applicability of rules, whereby a proposition $r_i$ is interpreted as "rule $r_i$ is applicable in the next step" (cases $\triangle$, $\diamond$) or as "rule $r_i$ has been applicable in the latest step" (case $\square$).

Please note that making decisions on post-conditions ($\square$) is *not equivalent* to making decisions on pre-conditions ($\triangle$, $\diamond$)! In Fig.3, for example, if $r_5$ has been applied we are sure that $r_1$ has been applied as well, but we are not sure that $r_3$ has been applied, too, if we only know that $r_7$ has been applied. A run of $I$ by $\mathcal{G}$ through $C$ from $\circ$ to $\odot$ is called a *graph transformation* on $G$.

It is quite obvious, by the way, that this control system would also be suitable for graph transformation rules with parameters $r_i(var_1, \dots, var_m)$ as known from systems like PROGRES [13] or FUJABA [5]. Further remarks on the interpretation and comparisons with other notions of graph transformation, which must be omitted due to lack of space in this paper, can be found in [7].

## 3.3 Implementation

At the moment [4], the control flow editor together with the control flow interpreter $I$ is almost completely implemented, except of unconditioned fork and join concepts $\top\top$ and $\bot\bot$ [7]. The implementation language is JAVA [12], and an API is provided for the sake of re-use. The integration of our control flow editor with the already existing graph grammar tool AGG [10] is basically done; full integration with AGG is ongoing work.

The GUI operations of the editor are *syntax-directed* such that the user is protected against making syntax errors while designing a control flow diagram $C$. (For example, it is not possible to construct a *goto* loop via the $\triangle$ item.) The rule names $r_i$ from a given graph grammar $\mathcal{G}$ can be inserted into $C$ via mouse & menu only, which also supports a consistent control flow design.

The control flow diagrams designed by the user are pretty-layouted automati-

[4] June 20, 2000

cally by the control flow editor. However, the tool also allows the user to manipulate the layout of a legal control structure via drag & drop for the sake of more sophisticated pictures. **Fig.4** shows a screen shot of the AGG system together with the control flow editor in the foreground.

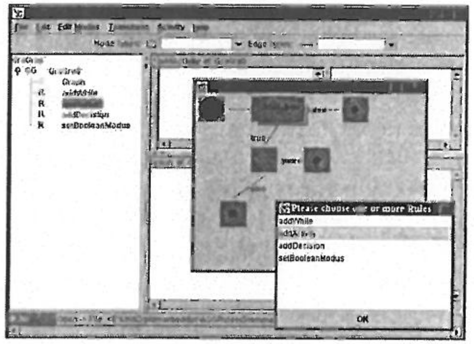

Figure 4: *screen shot of* AGG *with control flow editor*

# 4 Summary and Outlook

Graph grammars need to be enhanced with sophisticated control structures for several application purposes. Whilst other graph transformation systems offer either visual *or* explicit control structures, our approach of control flow diagrams supports the user in both visual *and* explicit control flow design. Our tool operates fully syntax-directed in order to protect the user from making avoidable mistakes. An operative run through a given control flow diagram with respect to a given graph grammar and a given host graph is called a graph transformation.

In the future, one could think of *hierarchical* control flow diagrams which contain further control flow diagrams in their activity items □. Provided with such hierarchical structures one could, for example, *switch* the supply of possibly applicable rules $r_i$ *on the fly*, similar to the concept of meta nodes in GRRR.

**Acknowledgments.** Thanks to our excellent programmer *Olga Runge*! At the moment, she is very busy with fully integrating the control flow editor into the existing AGG system...

# References

[1] C. Albayrak, *Die WHILE-Hierarchie für Programmschemata.* Ph.D.Dissertation, Faculty of Computer Science, RWTH Aachen, 1998.



Shaker Publ. Aachen/Maastricht, 1998

[2] E. Dijkstra, *GOTO Statement considered Harmful.* Comm. ACM 11/3, pp.147-148, 1968

[3] H. Ehrig et al. (Eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2 (Specifications and Programming).* World Scientific, Singapore 1999

[4] H. Ehrig et al. (Eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3 (Concurrency).* World Scientific, Singapore 1999

[5] T. Fischer et al., *Story Diagrams: a new Graph Rewrite Language based on the UML and JAVA.* H. Ehrig et al. (Eds.), TAGT'98 Workshop on Theory and Application of Graph Transformations. LNCS 1764, pp.296-309, Springer-Verlag, Berlin 2000

[6] M. Große-Rhode et al., *Modeling Distributed Systems by Modular Graph Transformation based on Refinement via Rule Expressions.* M. Nagl et al. (Eds.), AGTIVE'99 Workshop: Applications of Graph Transformations with Industrial Relevance. LNCS 1779, Springer-Verlag, Berlin 2000

[7] M. Kurt, *Entwurf und Implementierung einer Kontrollkomponente für attributierte Graphersetzung.* M.Sc.Thesis, Faculty of Computer Science, Techn. University of Berlin, 2000

[8] H.-J. Kreowski et al., *Nested Transformation Units.* International Journal on Software Engineering and Knowledge Engineering 7/4, pp.479-502, 1997

[9] G. Rozenberg (Ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1 (Foundations).* World Scientific, Singapore 1997

[10] AGG, http://tfs.cs.tu-berlin.de/agg/

[11] GRRR, http://www.cs.ukc.ac.uk/people/staff/pjr6/gdgr/main.html

[12] JAVA, http://www.javasoft.com/

[13] PROGRES, http://www-i3.informatik.rwth-aachen.de/research/progres/

[14] UML, http://www.omg.org/uml/ or http://www.rational.com/uml/

**Stefan Gruner** is now with the Laboratoire Bordelais de Recherche en Informatique, Université Bordeaux I, F-33405 Talence Cedex. His stay is generously granted by the TMR network GETGRATS. During the preparation of this contribution, he was with the Technische Universität Berlin. E-mail: stefan@cs.tu-berlin.de