Diagrammatic Control of Diagrammatic Structure Generation:

Additional Information to our DIAGRAMS 2000 Publication

Stefan Gruner
stefan_gruner@yahoo.de

Preface. This is an unpublished document, only available on demand. The purpose of this document is to provide some additional information on a paper that the author has published together with *Murat Kurt* for the first DIAGRAMS 2000 Conference in Edinburgh/Scotland, September 1^{st}-3^{rd}, 2000 [12]. As the scope of that conference was interdisciplinary, and as the text lenght of contributions has been restricted by the Editors of the according LNAI Volume, some interesting technical details could not have been mentioned in our original publication [12]. For this reason, a full version of our conference publication is provided with this document. Moreover: copies of our slides (presented at DIAGRAMS'2000) can be found in the Appendix of this document, too.

1 Introduction

Diagrams can be found at almost every place in a modern society, and the meaning of the term "diagram" is really vague. So, first of all, we have to declare what kind of diagrams we are working with, before we can start with the discussion of our ideas.

In **Fig.1**, three familiar examples of different kinds of diagrams are given. The example given in Fig.1**a**, which we may observe at a street crossing, belongs to the family of *sign*-like diagrams. The example given in Fig.1**b**, which we might watch on TV after a political election, belongs to the family of *function*-like diagrams. Those (and other possible) families of diagrams do *not* belong to the scope of our considerations in the following. Instead, we are interested in *graph*-like diagrams as depicted in Fig.1**c**, which may for example appear on public transportation maps. Here in the figure, the Central Station is connected with the University via the Main Street by tram line T1 and with the Air Port via the Music Hall by underground line U2. The Main Street and the Music Hall are additionally connected by bus line B12.

1.1 Universe of Discourse

The diagram given in Fig.1c is an example of a *labeled graph*. The labeled graphs constitute a subclass of the class of all diagrams and they are widely used in various disciplines of computer science and electrical engineering. For example, the well-known Entity-Relationship diagrams [3] in Database Schema Design,

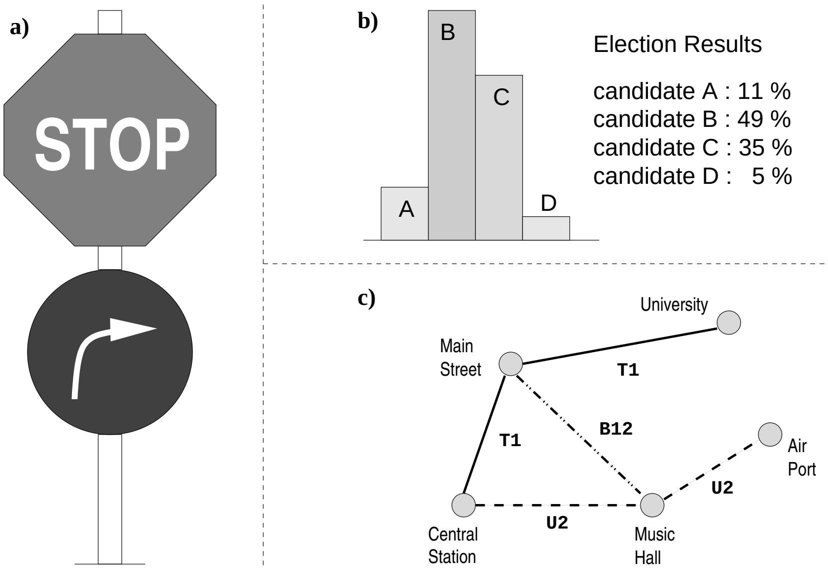


Fig. 1 *three examples of different kinds of diagrams*

Structured-Analysis diagrams [14] as well as Module- and Class Architecture diagrams [2][20] in Software Engineering, Petri nets [22] in Cybernetics, Automata- and Switching Theory, Semantic Networks [26] in Artificial Intelligence, or certain types of Circuit Maps [15] in electronic Hardware Engineering can sensibly be viewed as labeled graphs.[1]

Because of the technical relevance of such labeled graph diagrams, *graph grammars* (see below) have been developed as powerful and intuitive means for the generation and manipulation of such labeled graphs. Unfortunately, however, the operations of graph grammars are local and non-deterministic in principle so that not all the desired manipulation tasks can be fulfilled by graph grammars in an adequate manner. Thus, additional concepts of *control* are required. While textual control structures for the application of graph grammars are well-known already, more intuitive diagrammatic control mechanisms are still missing. We propose a combination of the UML *Activity Diagrams* and the earlier *Dijkstra Schemas* for this purpose.

Our main practical contribution is an (almost) already implemented *control flow editor* which the user can use to build up those diagrammatic control structures for the execution of graph grammar derivations. Moreover, our control

[1] Among the early literature on this topic we also can find a contribution wherein labeled graphs are —vice versa— *defined* as diagrams exclusively [16]. In our opinion, however, that early definition of diagrams does not properly fit the empirical phenomenon of diagrams in our urban society.

flow diagrams are *interpreted and animated* (by different colors) such that the the user can observe a diagram construction (or manipulation) running along the specified control flow diagram.

1.2 Organization of the Paper

The rest of our paper is structured as follows: In section 2, the notion of graph grammars is briefly introduced. (Readers who are familiar with graph grammars may omit that "auxiliary" section.) In section 3, we compare some related approaches to imposing control onto graph-grammatical diagram constructions. Section 4 explains the genealogy of our control flow diagrams which are described in section 5. Small examples are given in all these sections. In section 6, a bigger example is exercised for deeper intuition. Section 7 concludes our results.

2 Graph Grammars

Though the theory of graph grammars can be quite difficult in detail [23], the principle of graph grammars is so easy to understand that they are meanwhile applied as powerful tools supporting the solutions of several relevant problems in recent computer science [24][25]. In this section, the principle of *graph replacement* is briefly sketched which is necessary for understanding the context of our approach.

2.1 Example

Please imagine that the underground line U2 between the Music Hall and the Air Port (Fig.1c) has to be closed for the next season due to repair and modernization activities. For compensation, the transportation bureau of the town decides to prolong the bus line B12 from the Main Street to the Air Port. As a consequence, the transportation map must be adapted accordingly. Given the *graph replacement rule* of **Fig.2**, this adaption can easily be performed: First, we

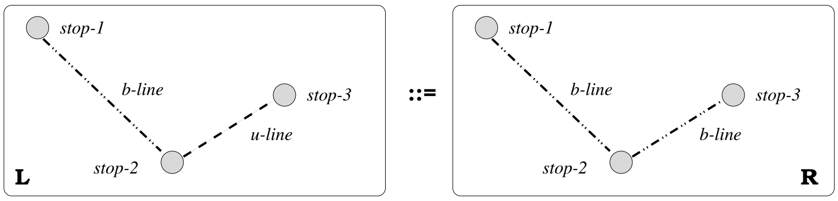


Fig. 2 *graph replacement rule*

have to find a *graph-homomorphic match* from the left-hand-side graph **L** (Fig.2) into the input graph (Fig.1c). We can match the *node variable stop-1* onto the node Main Street, the node variable *stop-2* onto the node Music Hall, and the node variable *stop-3* onto the node Airport. Then, we can match the *edge variable* (of bus line type) *b-line* onto the edge B12 and the edge type (of underground line type) *u-line* onto the edge U2 between Music Hall and Airport. Following the right-hand-side graph **R** of the rule (Fig.2), the matched underground line is *deleted*, and a new bus line is *created* which carries the same name as the bus line found in the pre-condition of **L**. In **Fig.3**, the application of the rule (from Fig.2) to the input graph (from Fig.1) is shown. Whilst the match from **L** to the input graph is the *pre-condition* of the graph replacement, an according match from **R** to the result graph is the *post-condition* of it.

2.2 Comment

The graph-homomorphic match described above is not the only possible one, because you could also match *stop-3* to Central Station and *u-line* to U2 between Music Hall and Central Station. The Advantages and Disadvantages of *that* kind of inherent non-determinism in the application of graph grammars are discussed in the literature and do not belong to the scope of this contribution — but a second kind of graph-grammatical non-determinism will be discussed below. Finally we have to state that, using graph grammars, it is not only possible to replace single edges (as done in our simple example), but also single nodes as well as rather complex subgraphs and patterns at once.

Concluding this "auxiliary" section we can state that a graph grammar is a *finite structure* $\mathcal{G} = \{G, r_1, \ldots, r_n\}$ whereby the start graph G is the axiom and the graph replacement patterns $r_1, \ldots, r_n$ are the rules of the calculus — which is a generalization of the well-known 1-dimensional (textual) Chomsky systems [4]. The different possible semantics —i.e. the different possible ways of *how to perform* an application of a graph replacement rule r_i in detail— are described in the literature [23] and do not belong to the scope of this paper.

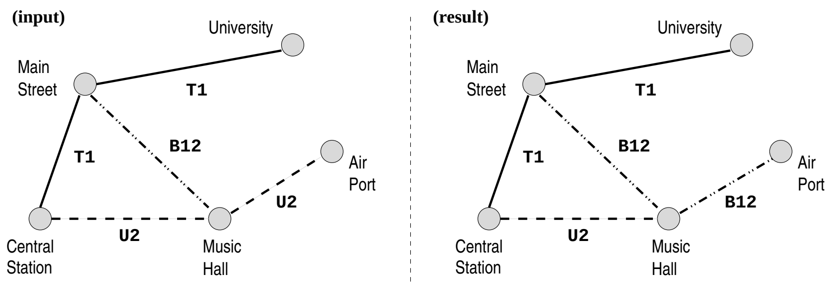


Fig. 3 *input and result of a rule application*

3 Comparison of related Control Approaches

After having briefly sketched the general idea of graph grammars in the previous section, we can now discuss some more specific issues which will directly lead us to the key issue of our paper. The question of non-determinism in the graph-grammatical generation of diagrams will be answered with special kinds of control diagrams for better support of diagrammatic structure generation.

3.1 Motivation of the Problem

As described in [24][25], graph grammars have turned out to be applicable for solving various problems of recent computer science. It is also known, however, that *pure* graph grammars are *no* suitable solutions for practical problems in several cases, because

i. graph grammar derivations (as any grammar derivations) are of non-deterministic nature whilst deterministic and predictable behavior is often required (see section 6 for example),

ii. graph grammar derivations are *local* operations on some data graph (input 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*. (Given a graph replacement rule $r : L \rightharpoondown R$, a data graph G and a match $m : L \to G$, the subgraph $m(L) \subseteq G$ is regarded as a neighborhood with respect to r.)

Example

For better readability of the public transportation map (Fig.1c), the transportation bureau decides to emphasize all junction stations where the passengers can step over from one transportation line to another one. This can be done with the graph replacement rule of **Fig.4**, but neither the rule itself nor the graph grammar which the rule belongs to is able to *ensure* that the rule is applied in *both* the neighborhoods of the station nodes Main Street *and* Music Hall for the sake of map-global consistency.

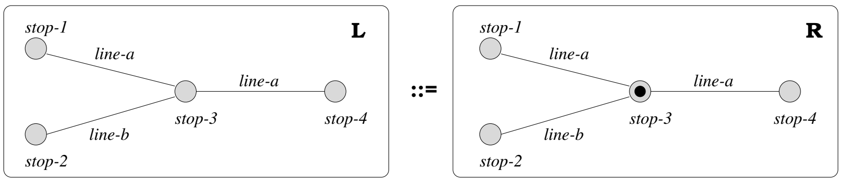


Fig. 4 *redrawing a junction of different lines: global application not ensured*

3.2 Solutions found in the Literature

For this reason, different concepts of *control* have been introduced in the recent graph grammar literature — in particular, we can mention the PROGRES system [31] and the GRRR system [29] which both are promising attempts to make the use of graph grammars feasible for various application tasks. Further approaches to imposing control on graph-grammatical diagram handling are, for example, described in [6] and [10]. Both the first approaches offer already implemented and usable tools, whilst both the latter approaches are still in a purely theoretical stadium as far as we know at the moment. A similar discussion is in [11].

Approaches without practical Evidence

The *transformation units* of [6] allow the specification of rule application sequences with ";" and application loops with "!". For example the expression $r_1!; (r_2; r3)!; r_4$ means to apply r_1 as often as possible, then to apply the sequence of r_2 and r_3 as often as possible, and finally to apply r_4 exactly once. Thus, the presented control flow language is only textual but not diagrammatic and, even worse, no concept of branching or case distinction is supported.

In the *rule expression* approach of [10], diagrammatic control structures are proposed the syntax of which is quite similar to the one of our approach. In the related approach, however, the semantics of the diagrammatic control symbols is slightly different. In contrast to the transformation unit approach of above, no concept of loop iteration is mentioned in the rule expression approach.

Approaches with implemented Tools

Comparing the control techniques of the relevant graph replacement tools PROGRES [31] and GRRR [29] we can observe that they are orthogonal to each other as shown in **Fig.5**:

PROGRES	GRRR	New Approach
explicit	*implicit*	*explicit*
textual	*diagrammatic*	*diagrammatic*

Fig. 5 *comparison of control concepts supported by relevant tools*

- Whilst the *explicit* control concepts of PROGRES are powerful they occur only in a *textual* notation which is rather inconvenient from a non-expert user's point of view. Explicit control in PROGRES means that the user can write program which executes the applicable graph transformation rules in the required manner.

- The opposit is true in GRRR: there we can find *diagrammatic* control concepts the handling of which is, therefore, quite user-friendly, but their control behavior is built-in, thus: *implicit*, and can not be modified by the user according to his requirements. 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 from outside.

3.3 Further Related Work to be mentioned

For the approach of *FUJABA* see [21]. From A.Schürr I received information that the idea of controlling graph grammar derivations by control flow diagrams has been published already before 1980 by H.Bunke (1972) and M.Nagl (1979), which is reported and theoretically worked-out by A.Schürr in his chapter of [23]. From G.Taentzer I received information that G.Engels and H.Ehrig speak about *Graph Procedures* in their approach to a module concept for graph transformation [8][9].

3.4 Our Approach

Starting from this discussion we wanted to combine the advantages and avoid the disadvantages of both the control concepts of PROGRES and GRRR which means that we required a controlling system of as well explicit and diagrammatic control structures.

As a result, we can present an (almost already) implemented diagrammatic *control flow editor* for the attributed graph grammar system AGG [28] as the main contribution of this paper. The control structures specified with our editor can be interpreted and executed, and the user can watch the system running on the *control flow monitor* [13]. The control flow monitor is another operational view of the control flow editor, and the conceptual basis of its *control flow diagrams* stem syntactically from a new combination of the widespread UML [2] and the earlier Dijkstra Schemas [5]. From a semantical point of view, however, the UML is left even farther behind in the interpretation of our special control flow diagrams.

At the end of this section it is worth mentioning that the process of constructing our control flow diagrams in the control flow editor can be described by graph grammars again such that —back to the beginning— the "methodological circle" is closed.

4 Watching out for suitable Control Structures

We have already mentioned above that, manipulating some data graph G by a graph grammar $\mathcal{G} = \{G, r_1, \ldots, r_n\}$, we have to face *two* different kinds of non-determinism, namely

i. the choice of *what* rule r_i shall be applied to G,
ii. the choice *where* a selected rule r_i shall be applied *within* the data graph G.

In the following, we are concerned with the first kind of non-determinism but *not* with the second one (the handling of which is sufficiently treated by the latest literature [7][21]). Thus: we want to be able to determine an application sequence $r(1); r(2); \cdots; r(m)$ which we call a *graph transformation* on G, where $r(i) \in \mathcal{G}$ for all $i = 1, \ldots, m$ and $0 \leq m \leq \infty$.

4.1 Dijkstra Schemas

The well-known *Dijkstra Schemas* [5] would allow us to specify the three most important control concepts in a diagrammatic fashion. These are: concatenation (*sequence*), conditioned decision (*if-then-else*), and conditioned iteration (*while*).

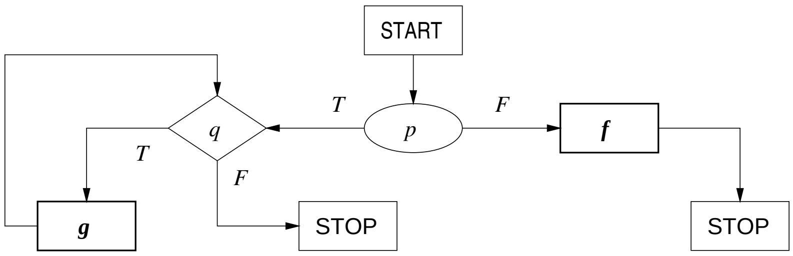


Fig. 6 *example of a Dijkstra Schema*

Example
The Dijkstra Schema depicted in **Fig.6** is interpreted in the following way: after the system has started, its global state is checked via some predicate p. If the result is false, some function f is executed and the system halts. Otherwise, some function g is executed until some predicate q results in false.

Comment
Unfortunately —due to certain restrictions concerning data types and functions [1]— we are not able to apply the elegant concept of Dijkstra Schemas immediately to our problem of diagrammatic graph transformation control [13]. The reason is that we are concerned with semi-deterministic operations on graphs whilst those *program schemas* are valid only for deterministic functions on ordinary global variables. Later we will see, however, what parts of the Dijkstra notion are re-used for which purpose in our hybrid control flow approach.

4.2 Activity Diagrams

Also the more recent *Activity Diagrams* [2] seem to offer us interesting diagrammatic features for the specification of control. Conditioned jumps are supported as well as the possibility of splitting the control flow into several concurrent branches.

Example

In **Fig.7**, a small example of an Activity Diagram is shown. After the system has started, some activity A is performed, followed by an activity B. If some condition *cond* is evaluated to false, the system jumps back to A and iterates. Otherwise, the both the activities C and D are performed in an asynchronous manner, this means: without respect to any application order in time. After both activities C and D have been applied, activity E is performed and the system halts.

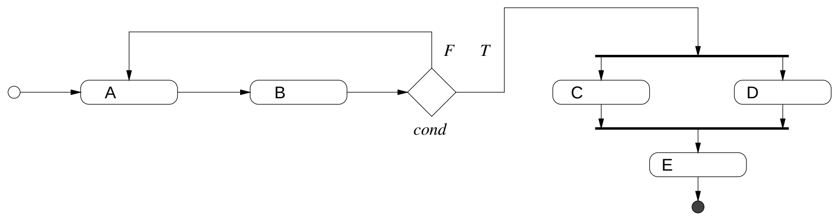


Fig. 7 *example of an Activity Diagram*

Comment

In contrast to the Dijkstra Schemas, Activity Diagrams are not restricted with respect to data types or function applications. Thus: as their original semantics is vague anyway, Activity Diagrams can easily be re-interpreted in such a way that our control requirements are fulfilled. Especially the *fork* and *join* items, the concurrent-asynchronous interpretation of which makes no sense in our case, will get a different meaning as shown in the following section. Please note further that some "legal" kind of non-determinism is re-introduced by the *fork* concept, which is not part of the strictly deterministic 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 [5]. Agreeing with Dijkstra at this point of discussion, we will avoid such "spaghetti" facilities in our approach as well.

4.3 Historical Note

As remarked in [27] (chapter 3.4, p.168)[2], the use of Flow Diagrams for the Control of Graph Grammar Derivations has already been suggested in [17].

5 Control Flow Diagrams and their Interpretation

In this section, we present the concepts of our solution to the problem how graph grammar operations on labeled data graphs can be controlled by diagrammatic hence intuitive and user-friendly means. We merge the most prominent concepts of Dijkstra Schemas and Activity Diagrams such that their advantages are increased and their disadvantages are decreased. Then we explain how our hybrid *control flow diagrams* [3] are interpreted in a graph transformation environment.

5.1 Structure

Our control flow diagrams —which belong to the diagram subclass of labeled graphs, too— are constructed from the following set of syntactical items. The definition can also be found in [13]

- All the nodes of the control flow diagram are connected by arrows $\rightarrow$ representing the control flow through the diagram.
- The arrows may (but need not always) be attached with the usual truth symbols T and F.
- There must be exactly one *start* symbol $\bigcirc$ and at least one *stop* symbol $\odot$. The start symbol has exactly one outgoing arrow, and the stop symbol has exactly one incoming arrow.
- The *rule application* symbol $\square$ must have exactly one incoming arrow and may have one (unlabeled) outgoing arrow; it may also have two outgoing arrows (which are labeled with T and F in this case).
- The *if-then-else* symbol $\triangle$ has exactly one incoming arrow and exactly two outgoing arrows which are labeled with T and F.
- Loops may be constructed only with the *while* symbol $\diamond$ (whilst the simulation of GOTO jumps with the *if-then-else* symbol is forbidden). The loop symbol must have exactly two incoming arrows and two outgoing arrows. One of the outgoing arrows is the exit. Every path beginning with the other outgoing arrow must eventually lead back to the $\diamond$ symbol such there is no other possibility of "illegally" leaving the loop without using the well-defined exit. The exit is labeled with F, the other outgoing arrow is labeled with T. Correspondingly, one of the loop inputs must come from outside of the loop and must not be a branch of it.

[2] Quotation: *"Schon in [Nagl79] wurde vorgeschlagen, Flußdiagramme zur Steuerung der Anwendungsreihenfolge von Graphersetzungsregeln zu verwenden."*

[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 hybrid Dijkstra-UML diagrams.

- Every rule application symbol □ must be attached with exactly one parameter r_i $(i = 1, \ldots, n)$.
- The decision symbols △ and ◇ must be attached with proposition-logical expressions (AND, OR, NOT) over some parameters r_i.
- Finally we may *fork* and *join* the control flow (without any condition) by using the according symbols ⊤⊤ and ⊥⊥, respectively.

5.2 Example

Fig.8 shows an example of how all these symbols may be used. The interpretation of our example of the control flow diagram $\mathcal{C}$ given in Fig.8 is explained below in an informal way.

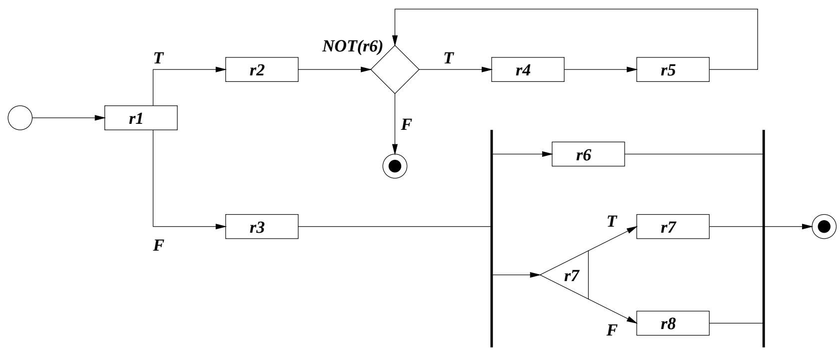


Fig. 8 *example of a control flow diagram $\mathcal{C}$ for some graph transformation*

5.3 Interpretation

Given a graph grammar $\mathcal{G} = \{G, r_1, \ldots, r_8\}$ the rules r_i of which shall operate on the data graph G. Given further an interpreter $\mathcal{I}$ which is able to operate on $\mathcal{C}$ and $\mathcal{G}$ by reading in $\mathcal{C}$ and writing in G according to the rules of $\mathcal{G}$. After having started in ◯, the interpreter *tries* to apply r_1. If this trial *has been* successful (*post-condition* T), $\mathcal{I}$ tries to apply r_2. Whether successful or not, $\mathcal{I}$ enters the *while* structure ◇. *As soon as* r_6 *is* applicable (*pre-condition* F), $\mathcal{I}$ stops at ⊙, 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), $\mathcal{I}$ tries to apply r_3 and enters the fork ⊤⊤ afterwards in any case. At this point, $\mathcal{I}$ makes a non-deterministic choice by chance. *Either* rule r_6 is tried or the *if-then-else* structure △ 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, $\mathcal{I}$ joins at $\perp\!\!\perp$ and stops the interpretation of $\mathcal{C}$ at $\odot$.

Comment

In general, the decision symbols $\triangle$ and $\diamond$ may be attached with proposition-logical expressions on propositions r_i, whereby a proposition r_i is interpreted as "rule r_i is applicable in the next step". In these cases, a label r_i appears as abbreviation of a boolean *predicate* expression $p(r_i)$ which is asking the applicability question. In the case of $\square$, however, the label r_i appears as abbreviation of a boolean *procedure call* expression $c(r_i)$ which returns true or false, and which has the possible *side effect* of a graph-transformative application of r_i.

Please note further that making decisions on post-conditions ($\square$) is *not equivalent* to making decisions on pre-conditions ($\triangle$, $\diamond$). In Fig.8, for example, if r_5 has been applied we are shure that r_1 has been applied as well, but we are not shure that r_3 has been applied, too, if we only know that r_7 has been applied.

A complete run of $\mathcal{I}$ by $\mathcal{G}$ through $\mathcal{C}$ from $\bigcirc$ to $\odot$ is called a *graph transformation* on G. Non-terminating loops over $\diamond$ are not regarded as graph transformations, thus: graph transformations are *finite per definitionem*.

The application environment of our diagrammatic control flow component is the attributed graph grammar tool AGG [28] which does not offer the possibility of concurrent application of graph replacement rules — nor do PROGRES or GRRR. Under these circumstances it does not make sense to interpret the $\top\!\!\top$ and $\perp\!\!\perp$ symbols in an asynchronous-concurrent manner as proposed by the UML. For this reason we have decided to interpret the *fork* construct as a non-deterministic (random) exclusive disjunction (XOR) which re-introduces the sometimes useful concept of "restricted anarchy".

Further remarks on the interpretation as well as comparisons with other notions of graph transformation can be found in [13]

5.4 Implementation

At the moment, our control flow editor together with the control flow interpreter $\mathcal{I}$ is almost completely implemented, except of unconditioned fork and join concepts $\top\!\!\top$ and $\perp\!\!\perp$ [13]. For reasons of re-usability, the implementation language is JAVA [30].[4] The integration and maintenance of our control flow editor with the already existing graph grammar tool AGG [28] is ongoing and future work.

The GUI operations of the editor are *syntax-directed* such that the user is protected from making syntax errors while designing a control flow diagram $\mathcal{C}$. The editor operations which the user can apply are formally specified by means of graph grammars again. The rule names r_i from a given graph grammar $\mathcal{G}$ can be inserted into $\mathcal{C}$ via "mouse & menu" only, which also supports a consistent control flow design.

[4] Of course we can provide the interested reader with the source code of our control flow editor. The tool should be re-usable in many JAVA-based software environments.

The control flow diagrams designed by the user are pretty-layouted automatically by the control flow editor. In this case, however, the tool also allows the user to manipulate the design via "drag & drop" for more sophisticated layout results. **Fig.9** shows the screen surface of our control editor in front of the AGG surface in the background.

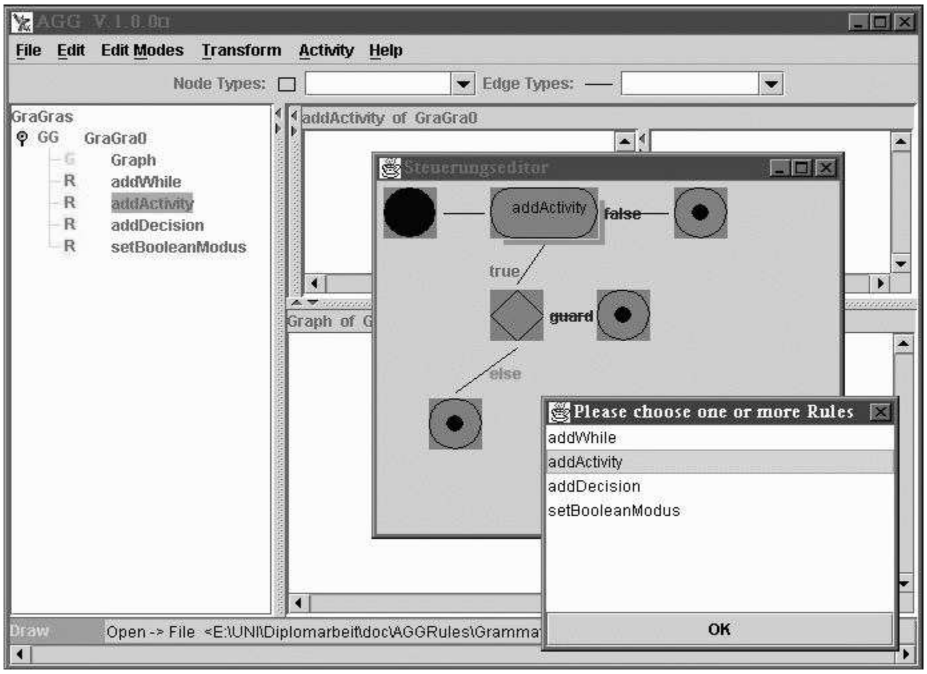


Fig. 9 *surface of the control flow editor with* AGG *in the background*

6 Example

In this section, a bigger example of our concepts is given for deeper intuition. We are constructing a public transportation map by use of some very simple graph replacement rules, and we show the control flow diagram by which the construction process is managed.[5] Our example graph grammar $\mathcal{G} = \{G, r_1, r_2, r_3, r_4, r_5\}$, which is depicted in **Fig.10**, consists of the following components:

[5] In graph-theoretical terms, the generated transportation map will be organized as a *spanning tree* of a graph.

G is the start graph. In contains a template of a town with 10 possible stations and several possible connections.

r_1 is a graph replacement rule which declares the destination and the first piece of tunnel of an underground line.

r_2 prolongs an underground line by one piece of tunnel between to stations. (The "active construction site" is identified by a bold-printed circle.)

r_3 creates a junction by connecting an underground station with a bus line.

r_4 prolongs a bus line into an empty area. (The "active construction site" is identified by a bold-printed circle again.)

r_5 wipes out a non-activated line pattern, (whereby the boxed circles serve as pattern for stations of any type, i.e. underground, bus, or junction).

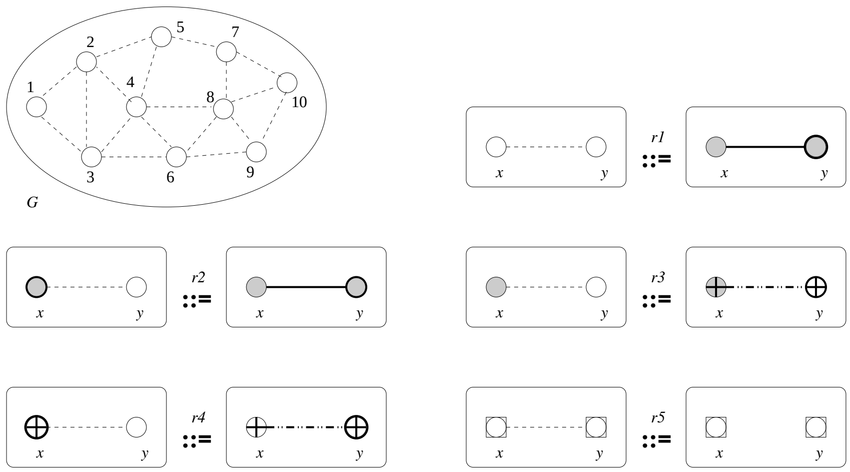


Fig. 10 *example graph grammar $\mathcal{G}$ with start graph G and five simple rules*

If we want to construct a useful transportation map with this graph grammar it is obvious that we must not apply the rules in a pure grammatical, i.e. non-deterministic manner. Otherwise for example, rule r_5 could destroy our map before any bus line or underground rail is drawn. Instead, we have to impose deterministic control onto the diagram construction process. Let's now assume that the public transportation bureau of our town has to satisfy the following planning constraints due to economic reasons:

α. There must be five underground stations.
β. The number of junctions shall be "small".
γ. Every possible place must be connected.

Provided with $\mathcal{G}$ and the control flow diagram $\mathcal{C}$ depicted in **Fig.11a**, we can construct a whole *family* of possible transportation maps [6] which all satisfy our economic constraints (the second of which is "soft"). The control flow diagram $\mathcal{C}$ implements the following diagram construction algorithm:

1. apply r_1 once;
2. apply $_2$ three times;
3. while r_3 or r_4 are applicable do:
   if r_4 is applicable apply r_4 once;
   else apply r_3 once;;
4. while r_5 is applicable do:
   apply r_5 once;;

The transportation map shown in Fig.11**b** is one possible result of our diagram construction procedure starting from G by $\mathcal{C}$ via $\mathcal{G}$. We hold the number of junctions comparatively small by giving r_4 a higher priority than r_3 in step *3.*

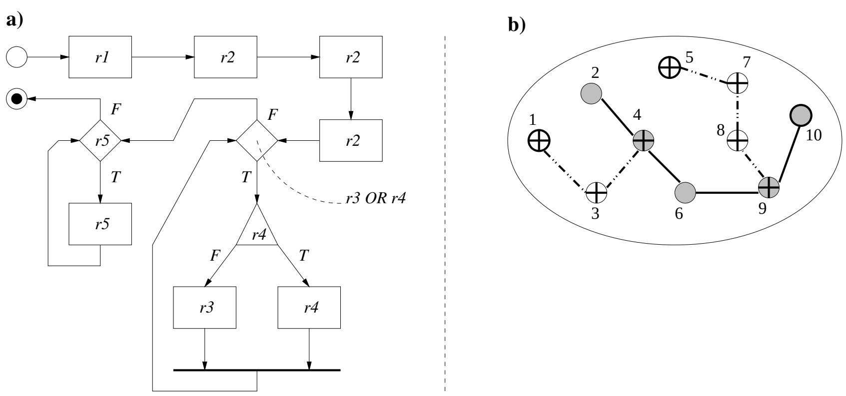


Fig. 11 *control flow diagram $\mathcal{C}$ and a transformation result from G*

7 Summary and Outlook

The class of labeled graphs is a relevant subclass of the class of all diagrams. Graph grammars are a powerful means for constructing and manipulating such graph-like diagram structures.

[6] This family of transportation maps is exactly the *language* $\mathcal{L}(\mathcal{G}_\mathcal{C})$ of our $\mathcal{C}$-restricted graph grammar $\mathcal{G}$. The advantage of multiple solutions is due to the non-determinism of type ii described in section 4.

However, pure graph grammars need to be enhanced with sophisticated control structures for several application purposes. Whilst other graph transformation systems offer either diagrammatic *or* explicit control structures, our approach of hybrid UML-Dijkstra-based control flow diagrams supports the user in both diagrammatic *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 data 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. Moreover, one could think of enhancing the control flow editor with sophisticated concepts of rule *parameterization* $r_i(var_1, \ldots, var_k)$ as it can be found in the PROGRES system.

References

1. C. Albayrak, *Die WHILE-Hierarchie für Programmschemata.* Doct.Diss., Fac. of Comp.Sc., RWTH Aachen, 1998. Shaker, Aachen/Maastricht, 1998
2. G. Booch et al., *The UML User Guide.* Addison Wesley, Reading 1998
3. P. Chen, *The Entity-Relationship Model: Towards a Unified View of Data.* ACM Transact. on Databs. Syst. 1/1, pp.9-36, 1976
4. N. Chomsky, *On certain Formal Properties of Grammars.* Inform. & Ctrl. 2/2, pp.137-167, 1959
5. E. Dijkstra, *GOTO Statement considered Harmful.* Comm. of the ACM 11/3, pp.147-148, 1968
6. F. Drewes et al., *Graph Transf. Modules and their Composition.* In [21]
7. H. Ehrig et al. (Eds.), *TAGT'98: 6th Internat. Worksh. on Theory and Applic. of Graph Transformation.* LNCS 1764, Springer, Berlin 2000
8. G.Engels et al., *Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems.* LNCS 1073, Springer, Berlin 1996
9. G.Engels et al., *Towards a Module Concept for Graph Transformation Systems: The Software Engineering Perspective.* Technical Report B-19, Universidad de les Illes Balears, Dptmnt. de Ciencies Matematiques i Informatica, Jun.1995
10. M. Große-Rhode et al., *Modeling Distr.Syst. by Modular Graph Transf. based on Refinement via Rule Expressions.* In [21]
11. S. Gruner et al., *A Visual Modeling Technique for Controlling Graph Transf.* Accepted for a Visualization-Workshop to the ICALP 2000 Conference. Published by Carleton Scientific, Waterloo 2000
12. S. Gruner et al., *Diagrammatic Control of Diagrammatic Structure Generation.* 1st Internat. Conference DIAGRAMS'2000 on the Theory and Applications of Diagrams. LNCS/LNAI, Springer, Berlin 2000
13. M. Kurt, *Entwurf und Implement. einer Kontroll-Komponente für attribut. Graph-Ersetzung.* M.Sc.Thesis, Fac. of Comp.Sc., Techn. Univ. Berlin, 2000
14. T. de Marco, *Structured Analysis and Syst. Specification.* Yourdon Pr., N.Y. 1978
15. G. de Micheli et al. (Eds.), *IEEE Transact. on Comp.Aided Design of integrated Circuits and Syst. 16/10.* IEEE Circuits & Syst. Soc., 1997

16. M. Nagl, *Formale Sprachen von markierten Graphen.* Doct.Diss., Arbeitsberichte des Instit. für mathematische Maschinen und Datenverarbeitung IMMD 7/4. Friedr.-Alex.-Univ. Erlangen, 1974
17. It must be either [18] or [19], unfortunately I could not manage to find out the literature exactly. *sg*
18. M. Nagl, *Graph-Grammatiken.* Vieweg-Verlag, Braunschweig 1979
19. M. Nagl, *GRAPL — a Programming Language for Handling Dynamic Problems on Graphs.* U. Pape (Ed.), WG'79 Proceedings of the 5th Workshop on Graph-theoretic Concepts in Computer Science. Hanser-Verlag, München/Munich 1979
20. M. Nagl (Ed.), *Building tightly integr. Softw. Developm. Environments.* LNCS 1170, Springer, Berlin 1996
21. M. Nagl et al. (Eds.), *AGTIVE'99 Worksh. on Applic. of Graph Transf. with Industrial Relevance.* LNCS 1779, Springer-Verlag, Berlin 2000
22. C. Petri, *Kommunikation mit Automaten.* Doct.Diss., Schriften des Instit. für instrumentelle Mathematik, Bonn 1962
23. G. Rozenberg (Ed.), *Handbook of Graph Grammars and Comp. by Graph Transf. 1 (Foundations).* World Scientific, Singapore 1997
24. G. Rozenberg et al. (Eds.), *Handbook of Graph Grammars and Comp. by Graph Transf. 2 (Specifications and Programming).* World Scientific, Singapore 1999
25. G. Rozenberg et al. (Eds.), *Handbook of Graph Grammars and Comp. by Graph Transf. 3 (Concurrency).* World Scientific, Singapore 1999
26. P. Winston, *Artificial Intelligence.* 3rd ed., Addison Wesley, Reading 1993
27. A. Zündorf, *Programmierte Graphersetzungssysteme: Spezifikation, Implementierung und Anwendung in einer integrierten Entwicklungsumgebung.* Deutscher Universitätsverlag, Wiesbaden 1996, ISBN 3-8244-2075-9
28. AGG, `http://tfs.cs.tu-berlin.de/agg/`
29. GRRR, `http://www.cs.ukc.ac.uk/people/staff/pjr6/gdgr/main.html`
30. JAVA, `http://www.javasoft.com/`
31. PROGRES, `http://www-i3.informatik.rwth-aachen.de/research/progres/`

Appendix: Slides

This appendix contains the slides of my talk given at DIAGRAMS 2000, the First Interdisciplinary Conference on Diagrams, Edinburgh (GB), August/September 2000.

This article has been processed with LaTeX in an LNAI compatible style, November 20, 2000.

Diagrammatic Control of
Diagrammatic Structure Generation

Stefan Gruner
stefan@labri.u-bordeaux.fr

Murat Kurt
murat@infonie.de

DIAGRAMS 2000
Internat. Conference
Edinburgh (GB)
Aug.-Sept. 2000

Support by Gabi Taentzer is gratefully acknowledged.

Topics

Graph Grammars

powerful, but not sufficient
for any Specification Purpose

Search for suitable Control Structures

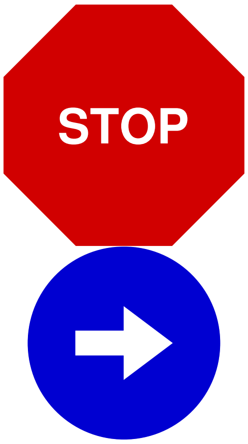
Dijkstra Schemas ?
Activity Diagrams ?

Editor - based Control

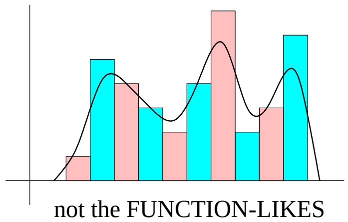
Java-based Tool for the AGG System

Preliminaries : Different Kinds of Diagrams

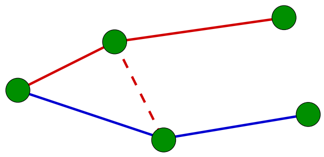
Which ones do we regard here ?



not the SIGN-LIKES



not the FUNCTION-LIKES

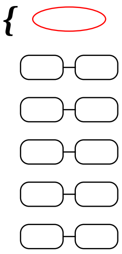


but the GRAPH-LIKES !

Preliminaries

Graph Grammars

***GG* = {**



}

A (usually finite) Start Graph G

A finite Set of finite
Graph Replacement Rules rJ

Two Kinds of Non-Determinism

a) Which Rule to choose for Application ?
b) Where to apply the choosen Rule in G ?

Graph Grammars in Diagrams Engineering

Non-Determinism

Obstacle in many Specification Tasks

b) Where to apply the choosen Rule in G ?

Solution: R U L E P A R A M E T E R S
with Influence on the Application Place in G
(Not Topic of this Talk)

a) Which Rule to choose for Application ?

Solution: C O N T R O L S T R U C T U R E S
transduce a Graph Grammar into a rule-based
Graph Transformation Program

Some Related Control Approaches

The PROGRES System
http://www-i3.informatik.rwth-aachen.de/research/progres/

Explicit Control _on_ Graph Transformation Rules
Control Structures in Textual Notation

The GRRR System
http://www.cs.ukc.ac.uk/people/staff/pjr6/gdgr/main.html

Implicit Control _in_ Graph Transformation Rules
Control Structures in Graphical Notation

Further Related Work [Bunke72] [Nagl79]
Graph Procedures [EhEn] Transformation Units [KuKr] FUJABA [Zuen]

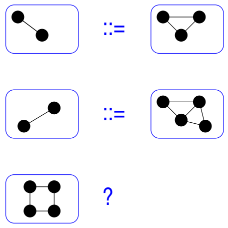
Principles of Related Control Concepts

PROGRES

production p1 : ::=

production p2 : ::=

query q : ?



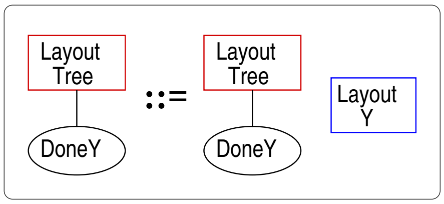
transaction t :

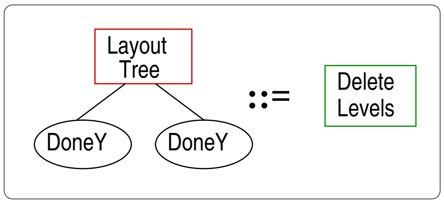
```
BEGIN
if q then p1 ;
loop p2 ;
END
```

Full Power of Computability,
but
Traditional Style of Programming

Principles of Related Control Concepts

GRRR





$\boxed{\text{Boxed}}$ Nodes represent the Activity of certain Sub-Routines.

Induced Priority Semantics is quite difficult to understand

Our Approach

Combine Advantages
Avoid Disadvantages

PROGRES	**GRRR**
Explicit	**Implicit**
Textual	**Visual**

New Control Flow Editor for the AGG System

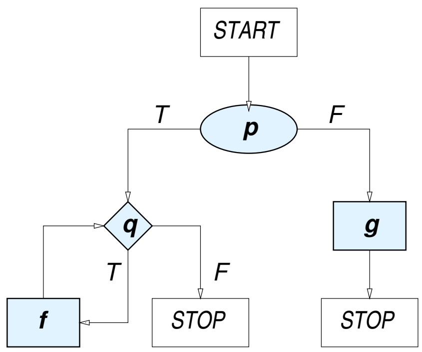
http://tfs.cs.tu-berlin.de/agg/

[AGG Statistics: Downloads from at least 26 different sites in 12 countries since November 1999]

Watching out for Suitable Control Graphs

Dijkstra Schemas ?

Functions - Predicates - If.Then.Else - While



Advantages

+ well formed
+ no Spaghetti

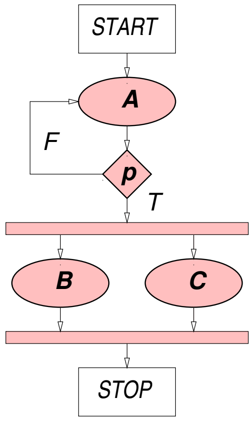
Disadvantages

- Concept of Functions not applicable to Graph Grammars

Watching out for Suitable Control Graphs

(UML) Activity Diagrams ?

Activities - Fork.and.Join - If.Then.Else



Disadvantages

- Badly Structured
- Spaghetti Jumps

Advantages

+ Activity Concept applicable to Graph Grammars
+ Restricted Non-Determinism possible (if needed)

Our Approach

Combine Advantages
Avoid Disadvantages

Dijkstra Schemas

Well Structured

Function Semantics

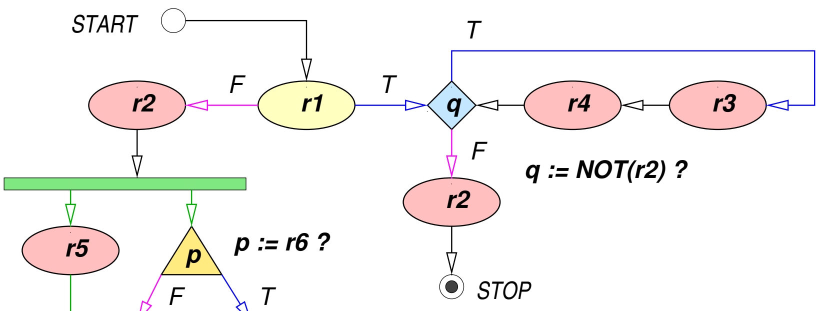
Activity Diagrams

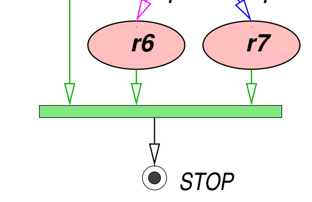
Spaghetti

"open" Semantics

Hybrid Control Flow Graphs
for the new AGG Control Flow Editor

Hybrid Control Flow Diagrams : Structure





Simple Activities
Activities with Fork on Implicit Postcondition
Fork on Explicit Precondition (if.then.else)
Unconditioned Fork and Join
While-Loop with Exit on Explicit Precondition
Simple logical Expressions over Names rJ

Hybrid Control Flow Diagrams Interpretation

Given a Graph Grammar GG with Start Graph G and Rules rJ

Simple Activity	rX	Try to apply Rule rX in G once. Continue in any case.
Activity with Implicit Post-condition	rY	Try to apply Rule rY in G once. Continue True-Line if Application has been successful. Continue False-Line else.
Fork on Explicit Precondition (if.then.else)	p	Test Applicability Condition p. Continue with True-Line, if p True. Continue with False-Line otherwise.
While-Loop on Explicit Precondition	q	Loop as long Applicability Condition q is True Exit Loop otherwise.
Uncondtioned Fork and Join		Continue with an arbitrary possible Line.

Hybrid Control Flow Diagrams : **Tool Support**

The User's View

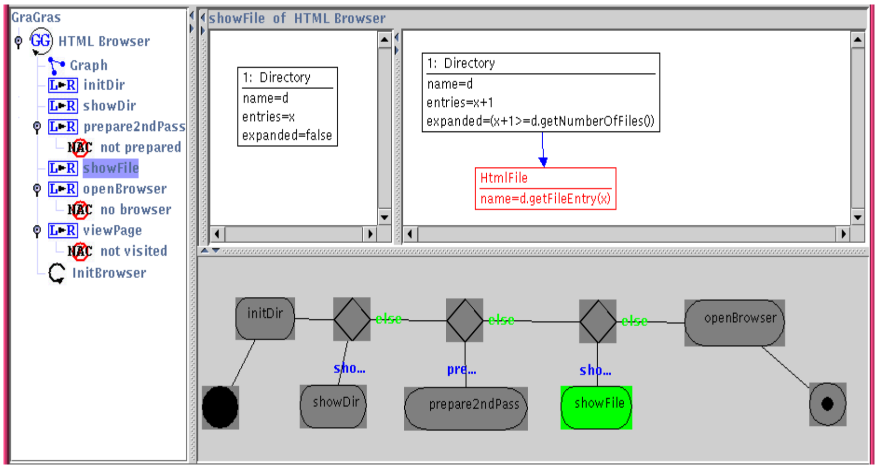
Syntax-Directed Construction of Control Flow Diagrams
with Respect to a given AGG Graph Grammar

Visible Interpreter-Mode Execution of Control Flow Diagrams
with current State highlighted and Graph Transformation Results
observable in the corresponding AGG Graph Transformation Window

Graph Transformation

A complete Run of the System through the Control Flow Diagram
is called a Graph Transformation.

AGG with Control Flow Editor **Screen Dump**



Programming by Olga Runge is gratefully acknowledged

Possible Extensions in Future

Introduction of Flow Parameters ?

Advanced Control by Input-Output-Variables
(similar to PROGRES)

Introduction of Hierarchical Control ?

Meta Control by Calling Sub-Control Graphs on the Fly
(similar to GRRR)

Summary

To make Graph Grammars feasible in Software Engineering
Control must be imposed in order to decrease Non-Determinism

Graph Grammars + Control = Programmed Graph Transformation

User-Friendly Control Concepts should be
Explicit as well as Visual and Well Structured

Tool Support of New Concepts should always be provided
and is provided by our implementation in fact

Many Thanks for Your Attention
Vielen Dank - Merci Beaucoup