

Visual Programming with Graph Rewriting Systems

A. Schürr, A. Winter

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

A. Zündorf[†]

Fachbereich 17 Softwaretechnik, Universität-GH-Paderborn
D- 33095 Paderborn, Germany
e-mail: zuendorf@uni-paderborn.de

Abstract

The multi-paradigm language PROGRES is the first rule-oriented visual language which has a well-defined type concept and supports programming with graph rewriting systems. To some extent, it has the flavor of a visual database programming language with powerful pattern matching and replacing facilities as well as backtracking capabilities. Until now, it was mainly used for specifying and rapid prototyping of abstract data types in software engineering environments. An integrated set of language-specific tools supports intertwined editing, analyzing, browsing, and debugging of specifications as well as generating prototypes in C (Modula-2) with Tcl/Tk-based user interfaces.

1 Introduction

Graphs play an important role within many areas of applied computer science, and there exists an abundance of visual languages and environments which have graphs as their underlying data model [24, 25, 26]. Furthermore, *rule-based languages* and systems haven proven to be well-suited for the description of complex transformation or inference processes on complex data structures. And *grammars*, for instance in the form of attribute grammars or definite clause grammars, are often used to describe syntax and semantics of textual languages and to generate parsers for them.

Although graphs and rule-based systems or grammars are quite popular among computer scientists, their combination in the form of graph rewriting systems or graph grammars is more or less unknown. At least one reason for this short-fall is that *graph grammar research* was from its very beginning in the early 70's focused on producing theoretical results, and working implementations based on these concepts were not available for a very long time. Furthermore, many people believe that modeling with graphs and graph rewriting systems leads to inherently inefficient implementations due to the NP-completeness of many graph algorithms. This situation changed gradually with the appearance of first graph rewriting system or graph grammar *implementations* like GraphEd [14], PAGA [12], and GOOD [11].

The essential idea of all implemented graph grammar or graph rewriting system approaches is to be a generalization of string grammars or term rewriting systems. The terms “graph grammar” and “graph rewriting system” are often

used as synonyms to each other. But strictly speaking, a *graph grammar* is a set of productions that generates a language of terminal graphs and produces nonterminal graphs as intermediate results. And a *graph rewriting system* is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs without distinguishing terminal and nonterminal results. Graph grammars are mainly used for synthesizing or recognizing graph-like data structures in biology, chemistry, and other research areas. Graph rewriting systems, on the other hand, are often used as *visual and executable specifications* of abstract data types or graph manipulating tools (cf. [5, 8]).

In the sequel we will present an integrated set of tools which supports **PRO**gramming with **GR**aph **RE**writing **S**ystems and which is available as free software. This system and its programming language PROGRES were already used

- for specifying tools and data structures of integrated software engineering environments [7],
- for describing process modeling, version control, and configuration management tools in CIM environments [21],
- as the underlying fundament of a new approach to diagram parsing [18],
- and finally for defining the semantics of a visual database query language [1].

PROGRES is a *visual programming language* in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. It was developed having the following *design goals* in mind:

- Use a graphical syntax where appropriate but do not exclude textual syntax when it is more natural and concise.
- Distinguish between data definition and data manipulation as database programming languages do and use graph class declarations to typecheck graph manipulating operations.
- Refrain users from the task to guarantee confluence of defined rewriting systems by keeping track of rewriting conflicts and backtracking out of dead-end derivations.
- Finally, do not rely on the rule-oriented programming paradigm for all purposes but support also imperative programming of rule application strategies.

Other rule-oriented visual languages like BITPICT [10], ChemTrains [2], or even VAMPIRE [17] do not meet all these requirements, as we will see later on in section 5.

This paper gives an informal overview of the language PROGRES and its programming environment. For all details concerning the language's formal semantics definition the reader is referred to [20].

[†] Supported by Deutsche Forschungsgemeinschaft 1990-1994 under contract number DFG Na 134/4-1, 4-2.

2 The PROGRES Language

PROGRES is a *strongly typed multi-paradigm language* with well-defined syntax and semantics based on graph rewriting systems. Being a mixed textual and diagrammatic language, it permits quite different styles of programming and supports

- description of graph schemata by means of a graphical as well as a text-oriented interface in a similar style as ER-oriented *database definition languages*,
- declaration of derived node attributes and derived binary relations like attribute grammars or *relational languages*,
- *rule-oriented* and diagrammatic specification of atomic graph rewriting steps by means of graph rewrite rules, and
- *imperative programming* of graph transformations by means of (non-)deterministic control structures.

Presenting PROGRES as a *visual language*, we will focus our main interest onto its diagrammatic parts, i.e. the specification of graph rewriting rules. Looking for an example that highlights this part of the language and demonstrates its *forward chaining* and *backtracking capabilities*, we came across the interesting problem of recognizing well-formed control flow diagrams, i.e. to recognize all those control flow diagrams that correspond to goto-less programs. We have to emphasize that the presented recognition algorithm is rather naive and not intended to be used in practice. It is subject of related research activities to develop new efficiently working graph grammar parsing algorithms [18].

Figure 1 shows on its left-hand side a well-formed control flow diagram. It consists of a while-loop with an if-statement as its body. The diagram on the right-hand side is not well-formed. It contains a control flow edge from the end of its else-branch (node 5) to the end of the program (node 7), and not to the end of the while-statement (node 6).

Both diagrams are examples of *directed, node and edge labeled graphs*. Node labels, in the following termed node

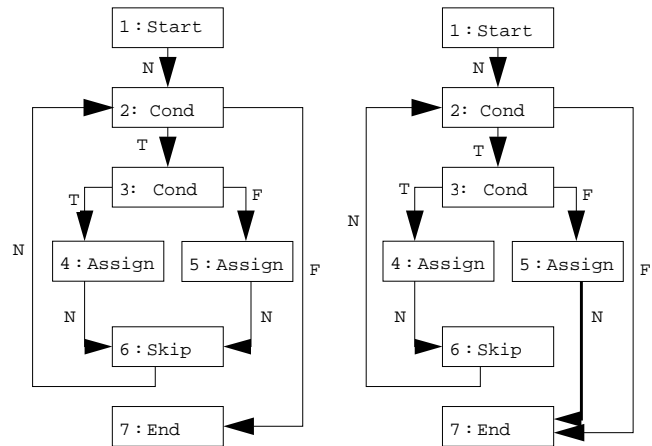


Figure 1: Well-formed & entangled control flow diagram

types, are used to distinguish nodes (objects) that play different roles within our control flow diagrams, like Cond(ition) or Assign(ment). Edge labels, termed edge types, distinguish edges with different meanings, as for instance T(rue) control flow, F(alse) control flow, and N(ormal) control flow. As we will see later on, some nodes have additional attributes which allow us to store unstructured graph properties. Cond and Assign nodes carry for instance a string-valued attribute with the corresponding piece of program text (see right-hand window of figure 2).

The specification of our parsing problem starts with the definition of a graph schema for the class of all allowed control flow diagrams. Afterwards, we will present a set of productions which return, applied to a well-formed control flow diagram, the empty graph and fail otherwise. The PROGRES screen dump in figure 2 displays a text-oriented as well as a graphical representation of the *control flow diagram graph schema* with three different categories of type declarations.

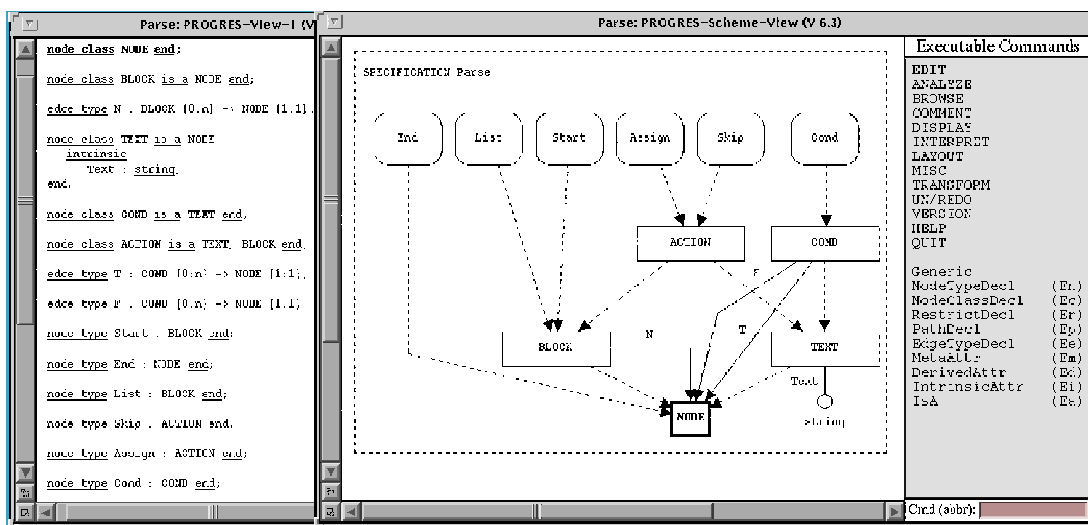


Figure 2: Textual and graphical editor parse views of a graph schema for control flow diagrams

Node and *edge type declarations* introduce labels for nodes and edges with common properties, whereas *node class declarations* are used to arrange sets of node types with common properties in the form of a multiple inheritance hierarchy. In this way, we are able to avoid redundant definition of node properties. The node type *Assign* belongs for instance to the class *ACTION* which is a subclass of the classes *TEXT* and *BLOCK*, and it inherits the following properties from these superclasses:

- Any node of type *Assign* has a string-valued *Text*-attribute which has a value of its own and is called “intrinsic” (instead of derived).
- Furthermore, all *Assign* nodes are sources of *N(ext)* edges that connect them with a single node of class *NODE*.

A significant difference between the *PROGRES* data model and usual object-oriented data models is that we distinguish between attributes and edges. This has the advantage that objects and *relationships* between objects may be *modeled separately* from each other and that we are not forced to “implement” edges by means of pairs of pointer attributes.

Therefore, declarations of edge types *T(rue)*, *F(alse)*, and *N(ext)* are denoted separately from their source and target node classes. These declarations contain cardinality constraints [1:1] for edge traversals in positive direction and [0:n] for edge traversals in reverse direction within these declarations. They impose the following *integrity constraints* onto all instances of control flow diagrams (cf. figure 2):

- Following *T*, *F*, or *N* edges starting at a selected source node results in one and only one target node.
- And traversing edges of one of these types in reverse direction results in an unconstrained large set of source nodes.

Therefore, we do not have to take diagrams into account where e.g. a *BLOCK* node is source of more than one *N* edge. This simplifies the specification of a correct and complete recognition algorithm considerably which consists of a small program (transaction) that controls the repetitive application of a set of *productions*. Each production has a left-hand and a right-hand side. Its application is divided into two phases:

1. Find a subgraph, termed *redex*, in a given host graph that matches the production’s left-hand side.
2. Replace the selected *redex* by a copy of its right-hand side, but preserve all those nodes and their context and attribute values which are shared among its left- and right-hand side.

The first production *RecognizeAxiom* in figure 3 has a left-hand (top) side with three nodes and two edges. It matches any subgraph in a host graph which consists of three different nodes of the required types connected by (at least) the required two edges and replaces the matched subgraph by the empty subgraph.

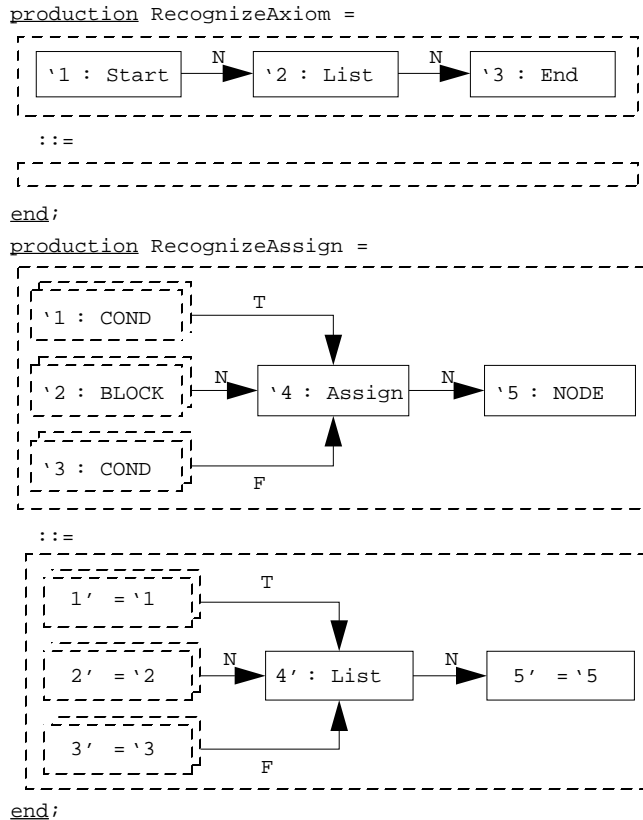


Figure 3: First examples of productions

The next production *RecognizeAssign* has a more complex form. It matches any *Assign* node in a host graph that is connected by an outgoing *N* edge to a node of a type of class *NODE*. Furthermore, it has a number of additional node-set-patterns (dashed double boxes) in its left- and right-hand sides. The first set-pattern with name *\1* matches all those *COND* nodes that are sources of a *T* edge with the selected *Assign*-node as target. In a similar way, set-patterns *\2* and *\3* match probably empty but in general arbitrarily large sets of *BLOCK* and *COND* nodes in the host graph, respectively. The production maintains all its matched nodes with their old attribute values and their incoming and outgoing edges (expressed by node inscriptions of the form *n' = \n*) with the exception of the selected *Assign* node and all edges in the production’s left-hand side. This node will be deleted together with all adjacent edges. Furthermore, a new *List* node will be created which inherits the outgoing *N* edge and all incoming *N*, *T*, and *F* edges of the old *Assign* node (all edges mentioned within the production’s right-hand side).

The production of figure 4 recognizes and removes a subgraph pattern that corresponds to an *if*-statement with two *List* node branches. These two nodes are not allowed to be target of another control flow edge as those mentioned explicitly in the diagram. The corresponding *application conditions* “*not 2ndInFlow*” are labels of double arrows pointing to the restricted nodes *\5* and *\6*. The restriction

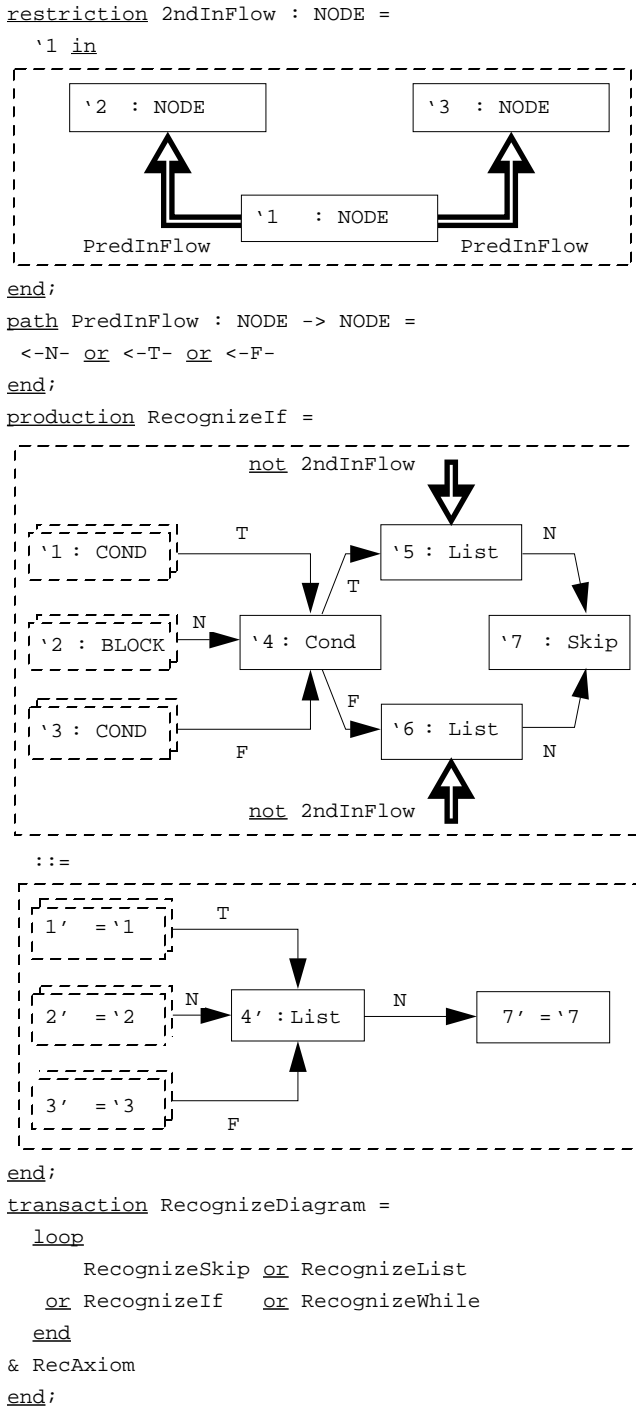


Figure 4: Production with application condition and control program for rewriting process

2ndInFlow itself is also defined in figure 4 and true for all those NODEs that are sources of at least two derived relations Pred(ecessor)InFlow. And the derived relation (path) Pred(ecessor)InFlow is defined to be the union of the inverse of N(ext), T(rue), and F(alse).

The productions for recognizing Skip- and While-statements as well as statement Lists are defined in a similar way

(see figures 6 and 7). Using these productions, we are able to write a program that recognizes all well-formed control flow diagrams. It consists of a loop, which executes as long as possible one of the productions listed in its body, followed by a call to the production RecognizeAxiom. All *rewriting conflicts* in the loop are resolved by selecting one production from the set of concurrently applicable productions and one of their possible matches in the host graph, randomly.

Regarding for instance the graph on the left-hand side of figure 1, we find two matching subgraphs for the production RecognizeAssign and one for the production RecognizeSkip. But note that applications of RecognizeSkip consume Skip nodes and thus prevent applications of Recognizelf afterwards. This a good example for locally reasonable graph rewriting steps that cause failure of the global rewriting process later on. In general, it is not possible to recognize wrong selections of productions or matches in advance. PROGRES has to *keep track of all selection possibilities and to back-track* out of dead-ends by undoing all effects of sometimes very long graph rewriting sequences and by selecting a remaining rewriting possibility.

In this way, the loop of transaction RecognizeDiagram selects applicable productions *nondeterministically* and reduces a control flow diagram step by step. Finally, all listed productions are no longer applicable, the loop terminates, and the following action after the ampersand, RecognizeAxiom, is activated. Its execution is either possible and a well-formed control flow diagram has been recognized, or back-tracking starts again and again until all possible derivation sequences within the loop have been tested and failed. In this case, the input was an “entangled” control flow diagram and the transaction RecognizeDiagram fails as a whole without any graph modifying effects.

To summarize, PROGRES is a *language for visually specifying* graph schemata, graph queries, and atomic graph rewriting steps. Additional language constructs, briefly mentioned at the beginning of this section, support the definition and maintenance of derived graph properties in a similar way as some database systems keep derived attributes or views in a consistent state [16]. And there exist even deterministic as well as nondeterministic control structures for programming complex graph transformation processes which either succeed and modify a given host graph or fail without any effects [22]. For further details about the language PROGRES and its formal semantics the reader is referred to [20].

3 The PROGRES Editor and Analyzer

The PROGRES editor and analyzer are those tools of our integrated programming environment which offer assistance for creating and modifying (hopefully) correct specifications. Both tools are tightly coupled and they are even integrated with the PROGRES interpreter of the next section. In this way, the tedious edit/compile/link/debug cycle is no longer

necessary and the environment's user is allowed to switch back and forth between editing, analyzing, and debugging activities. The *PROGRES editor* itself is not a monolithic tool but consists of a number of integrated subtools. These subtools support syntax-directed editing as well as text-oriented editing of specifications, pretty-printing (unparsing), and manual rearrangement of text and graphic elements. Figure 5 contains a rough sketch of all involved subtools and their data structures.

Syntax-directed editing works as follows: an active graphic window belongs to an active representation document that contains the currently selected representation element. This element in turn has a pointer to the corresponding portion of the specification's underlying abstract syntax tree, henceforth called logical increment, which is stored in a logical document. Both the logical document and the representation document are realized as *persistent attributed graphs*. Any logical document may have an arbitrary number of representation documents, which differ with respect to their layout, and any representation document may have an arbitrary number of associated graphic windows, which present different cut-outs of their underlying representation document.

Being aware of the *PROGRES* language's context-free syntax, the syntax-directed editor offers only those commands in a menu which *preserve context-free correctness*. The execution of a selected command starts with the modification of the logical document's abstract syntax tree. When all necessary updates of the logical document are completed, then a table-driven unparser propagates all modifications to all related representation documents.

In addition to syntax-directed editing the system also offers "*free*" text editing for the following reason: syntax-directed editing is preferred by users not familiar with a lan-

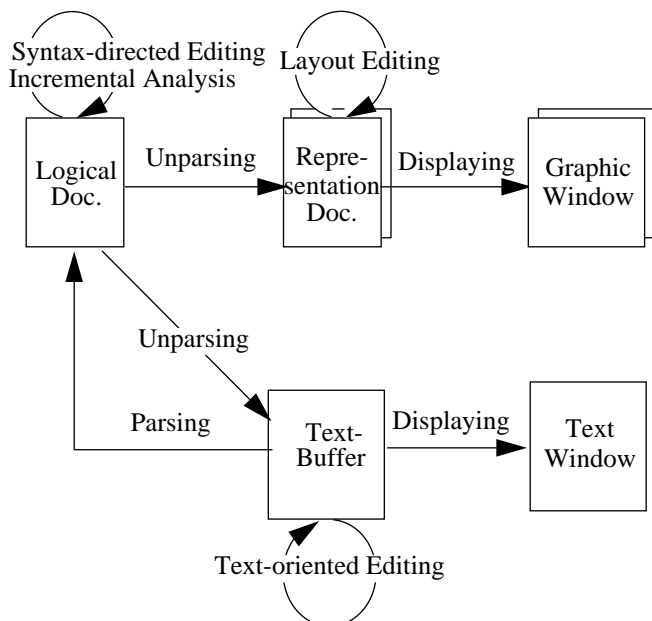


Figure 5: Editor data and transformation processes

guage's syntax, and it facilitates the creation of complex graphics considerably. But people tend to use text editors for restructuring textual parts of specifications and for entering expression-like constructs. Therefore, we decided to be liberal and to offer our users both styles of editing for all textual language constructs. It is subject of future work to include a "low-level" graphics editor together with a combined text and graphics parser into our environment (cf. [18]).

Text-oriented editing works as follows: An unparser translates the current increment into a line/column-oriented text representation that may be processed by a micro-emacs like text editor and is displayed in a text window (compare with figure 5). When text processing is finished, an LALR-parser checks the resulting piece of text for context-free correctness, calculates and executes a minimal sequence of necessary modifications of the logical document's abstract syntax tree skeleton. Finally, the same unparsing process as in the case of syntax-directed editing propagates logical document changes into related representations.

Before finishing this section, we have to explain *incremental analysis* in more detail. As already indicated, our analyzer is not always active but may be enabled and disabled on demand. The reason for offering these options is that even incremental analysis is sometimes too expensive to guarantee reasonable response times. Significant changes at a root class of a graph schema's inheritance hierarchy requires for instance reanalysis of a very large fraction of the corresponding specification document.

When being active, incremental analysis starts with a set of all affected abstract syntax subtrees in a logical document and determines all identifiers with modified bindings. When all identifier bindings are reestablished, an *acyclic dependency graph* is constructed that contains all directly or indirectly affected document subtrees which must be reanalyzed. Analysis itself processes one subtree after the other in the given dependency order and stores derived type-checking information in the form of additional edges and attributes as part of the internal attributed syntax graph realization of logical documents. In this way, our analyzer recognizes more than 350 different types of errors and it offers more or less detailed explanations in the form of error messages on demand.

The production *RecognizeSkip* of figure 6 contains four examples of typical *programming errors*:

1. The graph schema of figure 2 requires that *N* edges have *BLOCK* nodes as their sources. Therefore, *N* is not a proper label for an edge starting at the *COND* node '2 in the production's left-hand side (*COND* is not a subclass of *BLOCK*).
2. The second *N* edge within the production's left-hand side has the set-pattern node '5 as its target. But its declaration requires that any *BLOCK* node is connected to one and only one node of class *NODE*, i.e. we know that '5 matches always a single node and not a set of nodes.

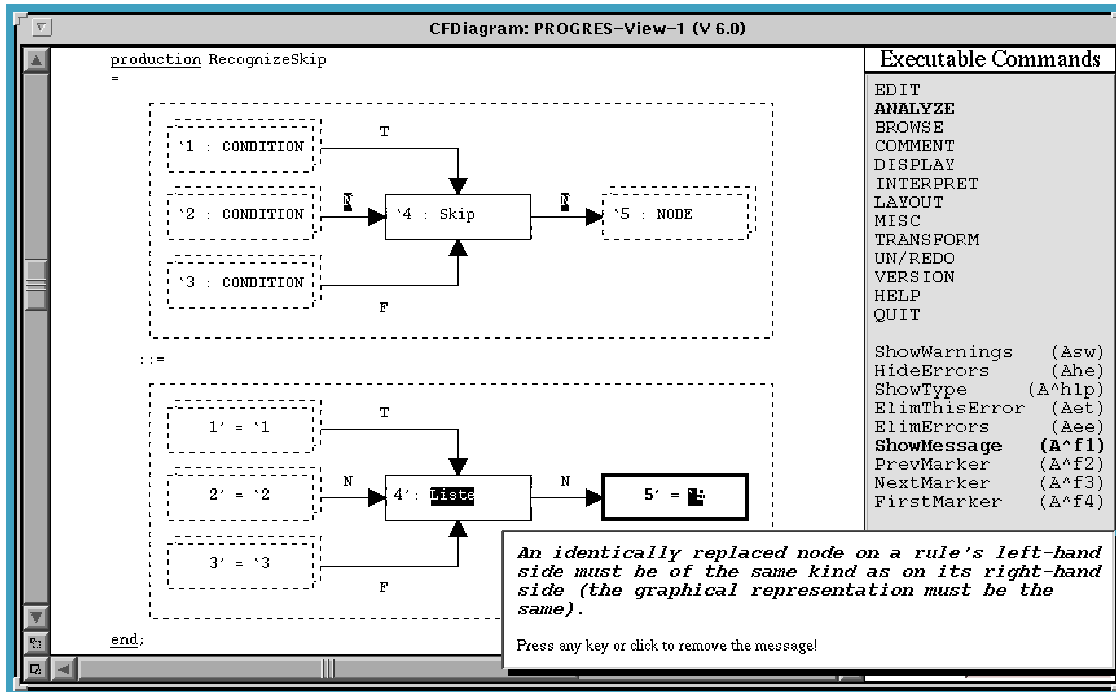


Figure 6: Production with four static semantics errors (increments with black background) and one displayed error message belonging to node 5':

Two errors concern the usage of edges in its left-hand side, and two concern the usage of nodes in its right-hand side.

3. The third error in the production's right-hand side is merely a typing error: the identifier Liste of node '4 does not have a corresponding declaration and must be replaced by the identifier List.
4. The last error is already explained in the displayed error message in the right/bottom corner of figure 6. It refers to the node 5' = '5 which is a "normal" node pattern on the right-hand side (recognizing a single node in the host graph) but is related to a set-pattern on the left-hand side (recognizing a set of nodes in the host graph).

To summarize, the tools presented within this section provide its users with substantial assistance for creating correct specifications. The hybrid syntax-directed as well as text-oriented editor prevents all violations of the PROGRES language's context-free syntax, and the analyzer highlights all errors with respect to the language's static semantics on demand.

4 The PROGRES Interpreter and Compiler

The PROGRES environment offers two alternatives how to animate a specification. The first one is based on *interpretation*. It is mainly used for debugging purposes, when intertwining of editing and execution activities is advantageous. The second one is to *compile* a specification into Modula-2 or C code. The generated code together with the environment's graph-oriented DBMS GRAS [16] and a Tcl/Tk-based user interface may be used as a *rapid prototype*. In both cases, the execution process is divided into two phases:

1. The initialization phase translates all graph schema declarations into an internal format for the underlying DBMS GRAS - including a dependency graph which is necessary for maintaining derived data in a consistent state.

2. Afterwards, arbitrary productions or transactions may be executed, which extend and transform the initially empty host graph step by step, and they may even be modified and recompiled while the execution process is running.

Figure 7 contains a snapshot of the recognition of the control flow diagram on the left-hand side of figure 1. The bottom window displays an already modified diagram where two Assign nodes are replaced by List nodes. Furthermore, a redex for the left-hand side of production RecognizeIf is already determined (bold-faced nodes in the diagram displayed as the lower part of figure 7).

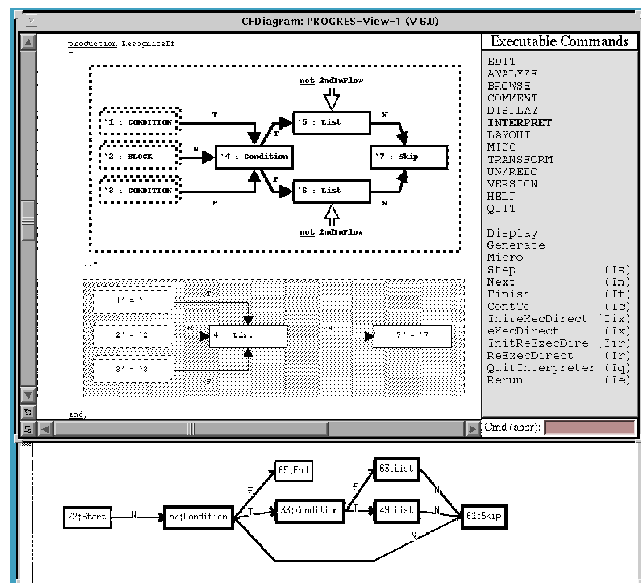


Figure 7: Applying a production to a given host

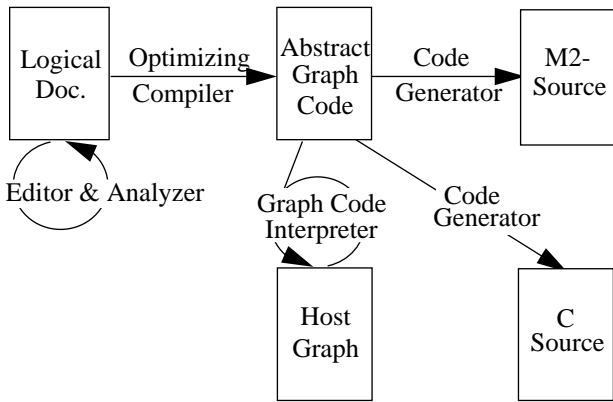


Fig. 8: Execution data and transformation processes

The production's right-hand side is the current interpreter increment. The next interpreter step deletes the selected Cond and List nodes with all attached edges. Afterwards, a new List node is created and embedded into the remaining diagram by drawing an N edge from node 57 to the new node and another N edge from the new node to node 61. Afterwards, a new production will be selected that recognizes another pattern within the resulting control flow diagram.

The diagram of figure 8 gives an overview of the main components and data structures which are involved in the above mentioned activities. It shows that the editor and analyzer of section 3 play the role of a conventional compiler's front-end and provide all information about a specification's underlying abstract syntax tree and its static semantics.

An *incrementally working compiler* takes this information as input and translates executable increments into intermediate abstract graph rewriting code. The code is stored within derived node attributes. The PROGRES system's internal attribute evaluation machinery is used for invalidating old pieces of code after editing activities and for recomputing code attributes on demand. The compiler is the most important component of the whole execution machinery and determines its efficiency to a great extent. This is especially true in the case of productions, where we have to choose between many different intermediate code sequences which all perform the required subgraph matching but differ considerably with respect to runtime efficiency [23].

The produced *abstract graph code* may be executed directly as input of an abstract graph rewriting machine. This machine combines the functionality of a conventional stack machine with graph rewriting operations and backtracking capabilities. It inherits from the afore-mentioned DBMS GRAS the ability to manipulate persistent graphs and, thereby, to store and resume debugging sessions, to maintain derived graph data in a consistent state, and to undo or redo sequences of graph modifications. Combining incremental (re-)compilation with undo&redo services, the PROGRES environment is even able to support the following debugging scenario:

1. Let us assume that the execution of a given specification eventually results in an erroneous host graph structure.
2. In that case, already performed graph rewrite steps may be undone or redone until the erroneous piece of code is identified.
3. Then, incorrect productions or transactions may be modified and (re-)analyzed as appropriate.
4. Afterwards, all modified increments will be recompiled on demand and the debugging session may be continued.

Beside interactive debugging, the PROGRES environment supports also the translation of specifications into equivalent readable Modula-2 or C source code. The generated code is even able to *backtrack*, which requires reversing a Modula-2 or C program's control flow, restoring old variable values, and undoing graph modifications. Using undo&redo services of GRAS, the main problem is to reverse a conventional program's flow of control without having access to the internal details of its compiler and runtime system. A description of the problem's solution is beyond the scope of this paper but may be found in [22].

5 Related Work

When comparing programming languages, it is often difficult to distinguish between properties of a language itself and properties of its accompanying set of tools. Therefore, we will not try to discuss related work about visual languages and about visual environments separate from each other.

Another problem for a comparison of PROGRES with other approaches stems from the fact that PROGRES is a *multi-paradigm language* (cf. beginning of section 2) which shares at least some properties with a large fraction of visual programming languages. In order to be able to keep the size of this section reasonable, we decided to concentrate on those visual languages which are either rule-oriented or offer at least *pattern matching constructs*, and which are built upon a more or less graph-like data model. Therefore, all so-called "graph rewriting" systems are omitted that are merely term rewriting systems which allow sharing of subterms and which have a text-oriented representation [13, 15].

Using these criteria, a surveyable number of visual rule-oriented languages remains for inspection. Compared with PROGRES their deficiencies come from the following sources: languages — like BITPICT [10], ChemTrains [2], or the recently presented underlying formalism of VIPR [4] — focus on *manipulation of data structures only* and data definition sublanguages are not provided. And even systems like VAMPIRE with its class hierarchies and icon rewriting rules [17] and PAGG with its node type definitions and graph rewriting rules [12] come without a rigid type concept and *without any type checking tools*. Therefore, all these systems postpone recognition of programming errors to runtime and suffer from the same disadvantages as any untyped or weakly typed programming language.

With respect to its graph schema definition capabilities, PROGRES is more similar to visual database programming languages [6, 11]. But these languages have less expressive pattern matching and replacing constructs (no set-patterns), and some of them are even not computational complete (no recursion). Furthermore, neither the above mentioned languages nor — to the best of our knowledge — any other visual rule-oriented language offers *nondeterministically* working control structures together with the ability to *back-track* out of dead-ends of locally failing rewriting processes.

6 Conclusion

This paper contains a brief presentation of the visual graph rewriting based language PROGRES and a more detailed discussion of its accompanying programming environment. The *PROGRES language* has a well-defined semantics and provides its users with graphical as well as textual constructs for the definition of graph schemata, derived graph properties, graph rewrite rules with elaborate pattern matching constructs, and complex graph transformation processes.

The *PROGRES environment* offers assistance for creating, analyzing, compiling, and debugging specifications as rapid prototypes of abstract data types. Being an *integrated set of tools* with support for intertwining these activities, PROGRES combines the flexibility of interpreted languages with the safeness of compiled and statically typed languages. A first version with about 400.000 lines of code is available as free software[†] for Sun workstations. A forthcoming new version supports even rapid prototyping by translating any given specification into a conventional C program with a Tcl/Tk based user interface.

It is subject of *future research activities* to add a module concept to the language PROGRES and to develop new compiler backends for generating C++ programs, which store their graphs either in main memory or in an object-oriented database systems (based on the ODMG-93 standard [3] of C++ bindings for OODBs).

7 References

- [1] Andries M., Engels G.: *Syntax and Semantics of Hybrid Database Languages*, in: [9]
- [2] Bell B., Lewis C.: *ChemTrains: A Language for Creating Pictures*, in: [26], 188-195
- [3] Cattell R.G.G. (ed.): *The Object Database Standard: ODMG-93*, San Mateo: Morgan Kaufmann Publ.
- [4] Citrin W., Doherty M., Zorn D.: *Formal Semantics of Control in a Completely Visual Programming Language*, in: [28], 208-215
- [5] Claus V., Ehrig H., Rozenberg G. (eds.): *Proc. Int. Workshop on Graph Grammars and Their Applications to Computer Science And Biology*, LNCS 73, Berlin: Springer Verlag (1978)
- [6] Czejdo B., Elmasri R., Rusinkiewicz M., Embley D.W.: *A Graphical Data Manipulation Language for an Extended Entity-Relationship Model*, in: IEEE Computer, vol. 23, no. 3, Los Alamitos: IEEE Computer Society Press (1990), 26-37
- [7] Engels G., Lewerentz C., Nagl M., Schäfer W., Schürr A.: *Building Integrated Software Development Environments Part I: Tool Specification*, in: acm Transactions on Software Engineering and Methodology, vol. 1, no. 2, New York: acm Press (1992), 135-167
- [8] Ehrig H., Kreowski H.-J. (eds.): *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Berlin: Springer Verlag (1990),
- [9] Ehrig H., Schneider H.-J.: *Proc. Dagstuhl-Seminar 9301 on Graph Transformations in Computer Science*, LNCS 776, Berlin: Springer Verlag (1994)
- [10] Furnas G.: *New Graphical Reasoning Models for Understanding Graphical Interfaces*, Proc. of CHI'91, 71-78
- [11] Gemis M., Paredaens J., Thyssens I., Van den Bussche J.: *GOOD: A Graph-Oriented Object Database System*, in: Buneman P., Jajodia S. (eds.): *Proc. 1993 ACM SIGMOD Conf. on Management of Data*, SIGMOD RECORD, vol. 22, no. 2, New York: acm Press (1993), 505-510
- [12] Göttler H., Günther J., Nieskens G.: *Use Graph Grammars to Design CAD-Systems*, in [8], 396-410
- [13] Glauert J.R., Kennaway J.R., Sleep M.R.: *Dactl: An Experimental Graph Rewriting Language*, in: [8], 378-395
- [14] Himsolt M.: *GraphED: An Interactive Graph Editor*, in: Proc. STACS 89, LNCS 349, Berlin: Springer Verlag (1988), 532-533
- [15] Kuchen H., Loogen R., Moreno-Navarro J.J., Rodriguez-Artalejo M.: *Graph-based Implementation of a Functional Logic Language*, in: Proc. ESOP '90, LNCS 432, Berlin: Springer Verlag (1990), 271-290
- [16] Kiesel N., Schürr A., Westfechtel B.: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*, in: Information Systems, vol. 20, no. 1, Oxford: Pergamon Press (1995), 21-51
- [17] McIntyre D.W., Glinert E.P.: *Visual Tools for Generating Iconic Programming Environments*, in: [25], 162-168
- [18] Rekers J., Schürr A.: *A Parsing Algorithm for Context-sensitive Graph Grammars (long version)*. Technical Report 95-05, Leiden University, 1995, available by ftp from ftp.wi.leiden-univ.nl, file /pub/CS/TechnicalReports/1995/tr95-05.ps.gz.
- [19] Schürr A.: *PROGRES: A VHL-Language Based on Graph Grammars*, in: [8], 641-659
- [20] Schürr A.: *Logic Based Structure Rewriting Systems*, in: [9], 341-357
- [21] Schwartz J., Westfechtel B.: *Integrated Data Management in a Heterogeneous CIM Environment*, in: Proc. CompEuro '93, Los Alamitos: IEEE Computer Society Press, 272-286
- [22] Zündorf A.: *Implementation of the Imperative/Rule Based Language PROGRES*, Technical Report AIB 92-38, RWTH Aachen, Germany (1992)
- [23] Zündorf A.: *A Heuristic Solution for the (Sub-)Graph Isomorphism Problem in Executing PROGRES*, Technical Report AIB 93-5, RWTH Aachen, Germany (1993)
- [24] Proc. of the 1989 IEEE Workshop on Visual Languages, Los Alamitos: IEEE Computer Society Press (1989)
- [25] Proc. of the 1992 IEEE Workshop on Visual Languages, Los Alamitos: IEEE Computer Society Press (1992)
- [26] Proc. of the 1993 IEEE Symp. on Visual Languages, Los Alamitos: IEEE Computer Society Press (1993)
- [27] lamitos: IEEE Computer Society Press (1992)
- [28] Proc. of the 1994 IEEE Symp. on Visual Languages, Los Alamitos: IEEE Computer Society Press (1994)

[†]see WWW page:

<http://www-i3.informatik.rwth-aachen.de/research/progres/>