Aachener Informatik-Berichte AIB 94-11

PROGRES, A Visual Language and Environment for PROgramming with Graph REwriting Systems

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

Abstract. *Graphs play an important role within many areas of applied computer science, and there exists an abundance of (diagrammatic) visual languages which have graphs as their underlying data model. Furthermore, rule-based languages and systems have proven to be well-suited for the description of complex transformation or inference processes on complex data structures. Although graphs and rule-based systems are quite popular, their combination in the form of graph rewriting systems (graph grammars) were more or less unknown among computer scientists for a very long time.*

Nowadays the situation is gradually improving with the appearance of a number of graph grammar based languages and tools. Currently, the multi-paradigm language PROGRES is the latest and most expressive descendant of a whole family of graph-rewriting system based specification languages. It has the flavor of a visual database programming language with powerful pattern matching and replacing facilities as well as backtracking capabilities. An integrated set of language-specific tools supports editing, analyzing, and debugging of applications, and even prototypes of PROGRES compilers with Modula-2 and C as target languages are available.

1. Introduction

Graphs play an important role within many areas of applied computer science, as e.g. in the form of

- data flow or control flow diagrams in compiler construction,
- entity relationship diagrams or semantic nets in database development,
- conceptual graphs or cliches in artificial intelligence research,
- and version or configuration graphs in software engineering and computer integrated manufacturing.

The list of examples presented above is by no means complete and there exists an abundance of visual languages and environments which have graphs as their underlying data model [7, 18, 19, 56, 57]. 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 string or tree language and to generate recognizers (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 [41, 45] 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. Nowadays the situation is gradually improving with the appearance of graph rewriting system (graph grammar) *implementations* like AGG [28], GraphED [22], PAGG [17], and our system PROGRES [36]. There exist even plans to combine the efforts of different research groups to develop a graph centered environment, called Grace, which encompasses at least the functionality of the systems AGG and PROGRES [25].

The essential idea of all implemented graph grammar or graph rewriting system approaches is to be a generalization of string grammars or term/tree rewriting systems, i.e. to replace the restricted data model of linear strings or (attributed) trees by the more liberal data model of (attributed) graphs. The terms "graph grammar" and "graph rewriting system" are often used as synonyms to each other. But strictly speaking, a *graph grammar* is a system of productions that generates, starting with a distinct axiom (start graph), a certain 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 etc., and graph rewriting systems - especially in the form of programmed graph rewriting systems [5, 49] - are often used as *visual and executable specifications* of abstract data types or graph manipulating tools (cf. [8, 13, 14, 15]. Early attempts to define graph manipulation languages may be found in [38, 32]. Currently, the multi-paradigm language PROGRES is the latest and most expressive descendent of a whole family of application-oriented graph rewriting languages [11, 12, 47]. It is a partly diagrammatic, partly text-oriented language that has already been used for

- specifying tools and data structures of integrated software engineering environments [12, 34] (its main application area),
- describing tools for process modeling, version control, and configuration management in CIM environments [50],
- defining the semantics of a visual database query language [1],
- and finally as the underlying fundament of a new approach to diagram parsing [43].

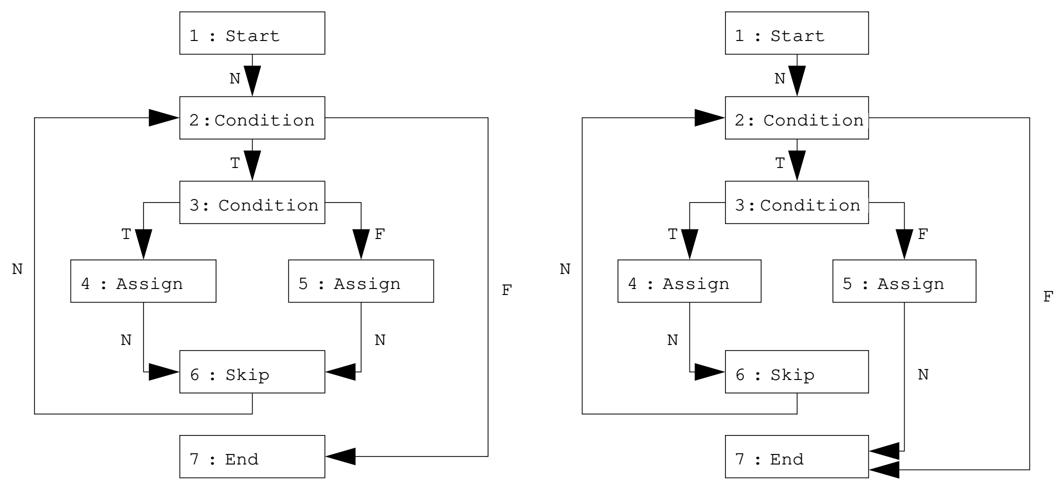
The remainder of this paper is organized as follows: Section 2 contains a brief introduction to (a subset of) the language PROGRES by means of a running example, recognizing well-formed control flow diagrams. Section 3 continues with an overview of the 400.000 lines of code large PROGRES environment including a discussion of its internal design and its underlying nonstandard database system GRAS. The following sections 4 and 5 present a discussion of the syntax-directed PROGRES editor with its textual/graphical user interface and its integrated incrementally working type checker as well as a description of the PROGRES interpreter and compiler that support interactive debugging and translation into Modula-2 or C source code. Finally, section 6 and 7 offer a comparison to related work and a summary of the current state of the PROGRES language and environment.

2. PROGRES Language and Running Example

PROGRES is a *strongly typed multi-paradigm language* with well-defined context-free syntax, type checking rules, and (dynamic) semantics. Its name is an akronym for *PRO*grammed *G*raph *RE*writing *S*ystems which are the language's underlying formalism and are in turn defined by means of nonmonotonic logic and fixpoint theory [48]. Being a mixed textual and diagrammatic language, it permits quite different styles of programming and supports

- description of graph database schemes by means of an ER-like as well as a text-oriented interface,
- declaration of derived graph properties by means of arithmetic expressions for derived node attributes, and by means of so-called path expressions as well as complex graph patterns for derived node sets and relations,
- rule-oriented and diagrammatic specification of atomic graph rewriting steps by means of parametrized graph rewrite rules (productions) with complex preconditions, and
- imperative programming of composite graph transformation processes by means of deterministic and nondeterministic control structures.

Presenting PROGRES as a kind of *visual programming 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 (parsing) all well-formed control flow diagrams in the class of all control flow diagrams, i.e. to recognize all those control flow diagrams that correspond to goto-less programs[1].



a) Well-Formed Control Flow Diagram b) Entangled Control Flow Diagram

Fig. 1: Examples of control flow diagrams.

1) Parsing diagrams is an example for programming with PROGRES which is easy to understand and demonstrates the language's graph rewriting capabilities very well. Nevertheless, we do not claim PROGRES being a competitor to those (visual) diagram syntax description languages which are especially tailored to this application domain [21, 31, 27].

Figure 1 contains on its left-hand side an example of a well-formed control flow diagram that consists of a while-loop with an if-statement as its body. The second diagram on the right-hand side of figure 1 is not well-formed. The N(ext) control flow from the end of the if-statement's else-branch (node 5) leads to the end of the program (node 7) and not the end of the while-statement's body (node 6).

Both diagrams are examples of *directed, node and edge labeled graphs*. Node labels, in the following termed node types, are used to distinguish nodes (objects) that play different roles within our control flow diagrams, like Conditions, Assignments etc.; and edge labels, termed edge types, distinguish edges (binary relations) with different meanings, as for instance T = True control flow, F = False control flow, and N = Next/Normal control flow. As we will see later on, some nodes have additional *attributes* which allow us to store unstructured graph properties. Conditions and Assignments 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 (database) scheme 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[2]. The PROGRES environment screen dump in figure 2 displays a text-oriented as well as an ER-like representation of the *control flow diagram graph scheme* with three different categories of type declarations. *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.

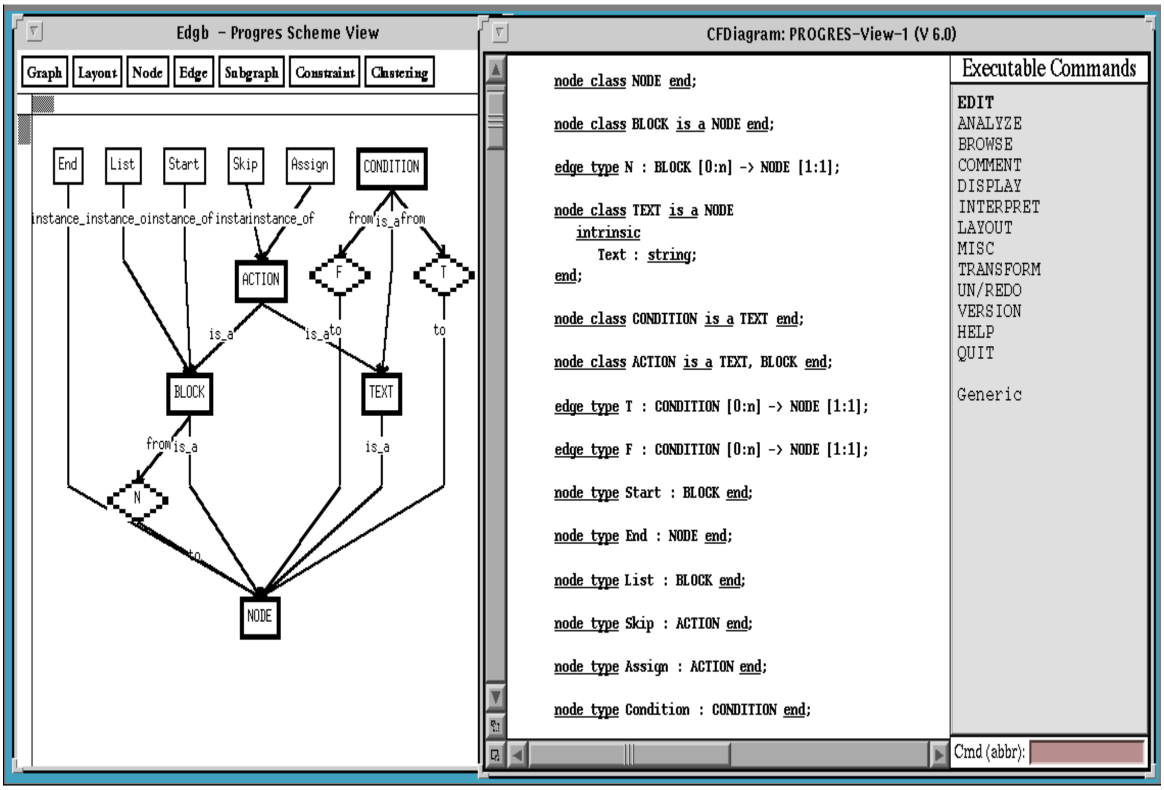


Fig. 2: Graph scheme for control flow diagrams.

2) Failure is guaranteed for scheme-consistent graphs, only (see also next page).

In this way, we are able to avoid redundant definition of node properties[3]. 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 therefore called intrinsic (instead of derived).
- Furthermore, all Assign nodes are sources of N(ext) edges that connect them with one and only one node of class NODE.

The presented type concept is similar to the data model of the ODMG-93 object database standard proposal [6]. Abstract types describe properties (interfaces) of objects, and abstract types, which have a default implementation, are termed (node) classes. Furthermore, any object type may have an arbitrary number of implementations, which are termed node types in the case of PROGRES. Thus any object (node) belongs to a uniquely defined implementation (node type) which belongs in turn to an abstract type (class) description and, thereby, to all its supertypes (superclasses). This constitutes an *object/type universe with three layers*, where node instances as well as node types are typed first-class objects that may be used as parameter or variable values. As a consequence PROGRES offers its users subtype and parameter polymorphy for writing reusable specification fragments.

A significant difference between the PROGRES data model and almost all object-oriented data models is that we distinguish between node attributes and edges. This has the advantage that objects and *relationships* between objects may be *modeled separatel*y from each other and that we are not forced to "implement" edges by means of pairs of pointer (lists) attributes, thus avoiding the problem of dangling references and facilitating the extension of already existing class hierarchies significantly [23]. 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:

- Following T(rue), F(alse), or N(ext) 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(ext) 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 and its application is divided into two phases:

(1) Search for 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.

---

3) But we have to deal with new problems like inheritance conflicts; cf. [12, 47] for informations about class hierarchies, inheritance of attribute evaluation rules etc.

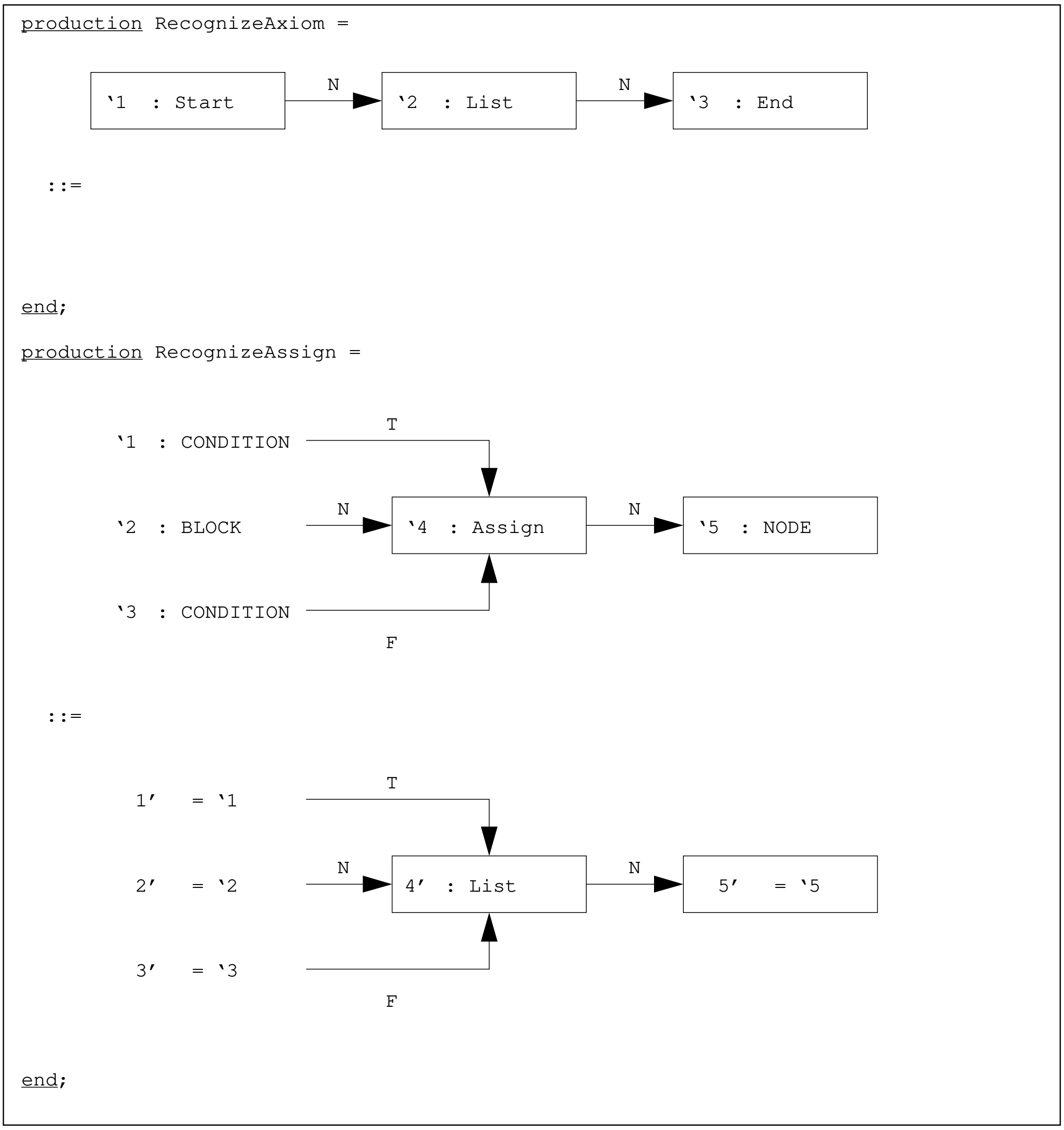


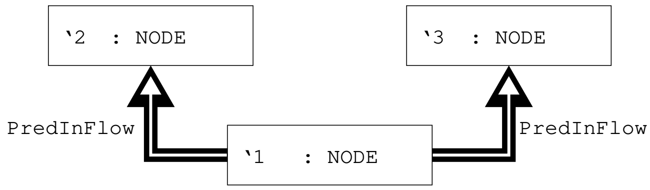
Fig. 3: First examples of productions.

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.

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(ext)-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 CONDITION nodes that are sources of a T(rue) edge with the selected Assign-node as target. In a similar way, set nodes ‘2 and ‘3 match probably empty but in general arbitrarily large sets of BLOCK and CONDITION 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 in its right-hand side) with the exception of the selected Assign-node and all explicitly mentioned edges in the production's left-hand side. This node will be deleted together with all adjacent edges, and a new list node will be created which inherits the outgoing N(ext) edge and all incoming N(ext), T(rue), and F(alse) edges of the old Assign node (all edges mentioned within the production's right-hand side).

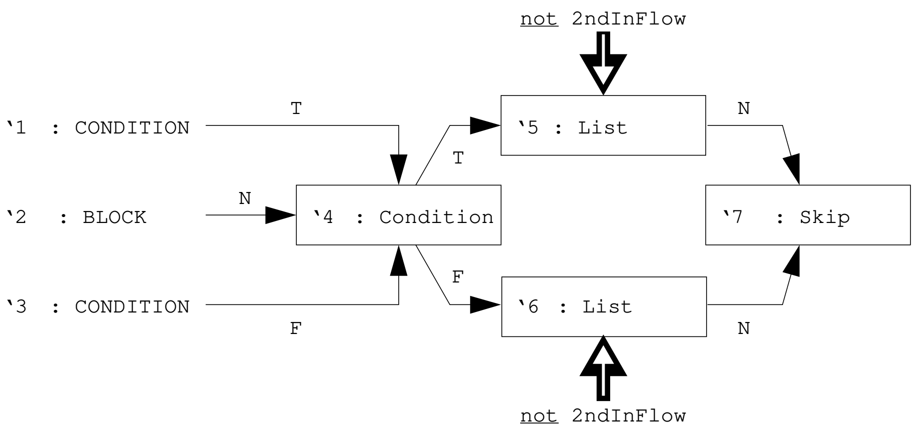
```
restriction 2ndInFlow : NODE =
  `1 in
```



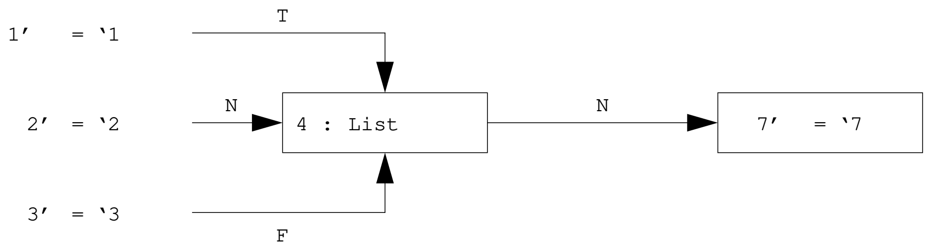

```
end;

path PredInFlow : NODE -> NODE =
   <-N- or <-T- or <-F-
end;

production RecognizeIf =
```




```
  ::=
```




```
end;

transaction RecognizeDiagram =
  loop
    RecSkip
   or RecList
   or RecIf
   or RecWhile
  end
 &RecAxiom
end;
```

Fig. 4: Production with application condition and control program.

The production of figure 4 recognizes and removes a subgraph pattern that corresponds to an <u>if</u>-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* "<u>not</u> 2ndInFlow" are labels of double arrows pointing to the restricted nodes '5 and '6. The restriction 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 also figures 7 and 8). Using these productions, we are now able to write a small 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 out of 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 RecognizeIf 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 but PROGRES has to *keep track of all selection possibilities and to backtrac*k 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 selects applicable productions *nondeterministically* and reduces a given 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 backtracking 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 whole transaction RecognizeDiagram fails as a whole without any graph modifying effects.

To summarize, PROGRES is a *multi-paradigm language for visually specifying* graph schemes, 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 (attributes, binary relations, and node sets) in a similar way as in attribute grammars [26, 44]. 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 (atomic transactions). For further details about the language PROGRES and its formal semantics the reader is referred to [47, 48].

3. The PROGRES Environment - An Overview

Equipped with some basic knowledge about the language PROGRES, the reader probably can imagine the difficulties we had to face during the development of its integrated programming environment. Nevertheless, a 400.000 lines of code large prototype is now available for Sun workstations[4] and consists of the following *integrated set of tools*:

- a mixed textual/graphical syntax-directed editor together with an incrementally working table-driven pretty-printer, called unparser, and a layout editor,
- an integrated (micro-)emacs like text editor together with an incrementally working LALR-parser,
- an incrementally working type-checker which detects all inconsistencies with respect to the PROGRES language's static semantics and explains highlighted errors by means of more than one hundred messages,
- an import/export interface to plain text editors and text-processing systems (currently, for text processing system FrameMaker from Frame Technology Corporation only),
- a browsing tool for finding declarations or applied occurrences of given identifiers, highlighting yet incomplete specification parts or unused declarations, and for displaying inferred type information,
- a browser which is an instantiation of the generic graph browser EDGE [37] from the university of Karlsruhe[5]and displays graph schemes in an ER-like fashion,
- an integrated interpreter which translates PROGRES specifications incrementally into intermediate code and forwards this code to an abstract graph rewriting machinery,
- a graph-browser for monitoring host graphs during an interpreter session,
- two cross-compilers which translate PROGRES intermediate code into plain Modula-2 and C code, and finally
- tools for version management and three-way merging of different versions of one specification document (for further details see [52, 53]).

When developing all these tools we had the following major *requirements* for their functionality in mind, which are "state of the art" for integrated programming environments or environment generators like IPSEN [35], CPSG [44], CENTAUR [4], GARDEN [42], and others:

- *Structure-oriented*: (almost) all tools operate on syntactical units, termed logical increments, of specification documents.
- *Command driven*: all tool activities are started by selecting commands in a menu (by either using the mouse or self-defined key stroke sequences), and the menu offers at any time only reasonable commands for the currently selected specification increment.
- *Interactive*: all effects of selected commands will be computed immediately and reported to the user when requested, as e.g. all violations of the PROGRES language's static semantics rules.
- *Incremental*: the costs for executing a command are (as far as possible) proportional to the size of the initiated document modification and not to the size of the document itself.

---

4) Via anonymous ftp from ftp.informatik.rwth-aachen.de in directory pub/unix/PROGRES.

5) Will be replaced soon by an own version of a graph scheme editor.

- *Integrated*: all tools of the environment cooperate with each other and support intertwining of different activities like editing, analyzing, and debugging specifications.
- *Persistent*: all effects of commands are immediately persistent, survive even host system crashes, but may be undone and redone even after the execution of an arbitrary number of subsequent commands.
- *Multi-view-oriented*: cutouts of specification documents may be displayed in an arbitrary number of overlapping windows with different scaling factors, layout, and even varying style of presentation (textual, diagrammatic).
- *Uniform*: all tools possess the above mentioned properties and have a uniform user interface.

Before discussing the functionality and realization of the environment's most important tools in section 4 and 5, we will first explain how we were able to fulfill the requirements listed above. For this purpose, we should have a look onto the environment's architecture in figure 5. The presented diagram consists of six layers of subsystems on top of the operating system arranged into three columns, where each subsystem uses resources of other subsystems on the same layer or the layer beneath it and introduces a new level of abstraction:

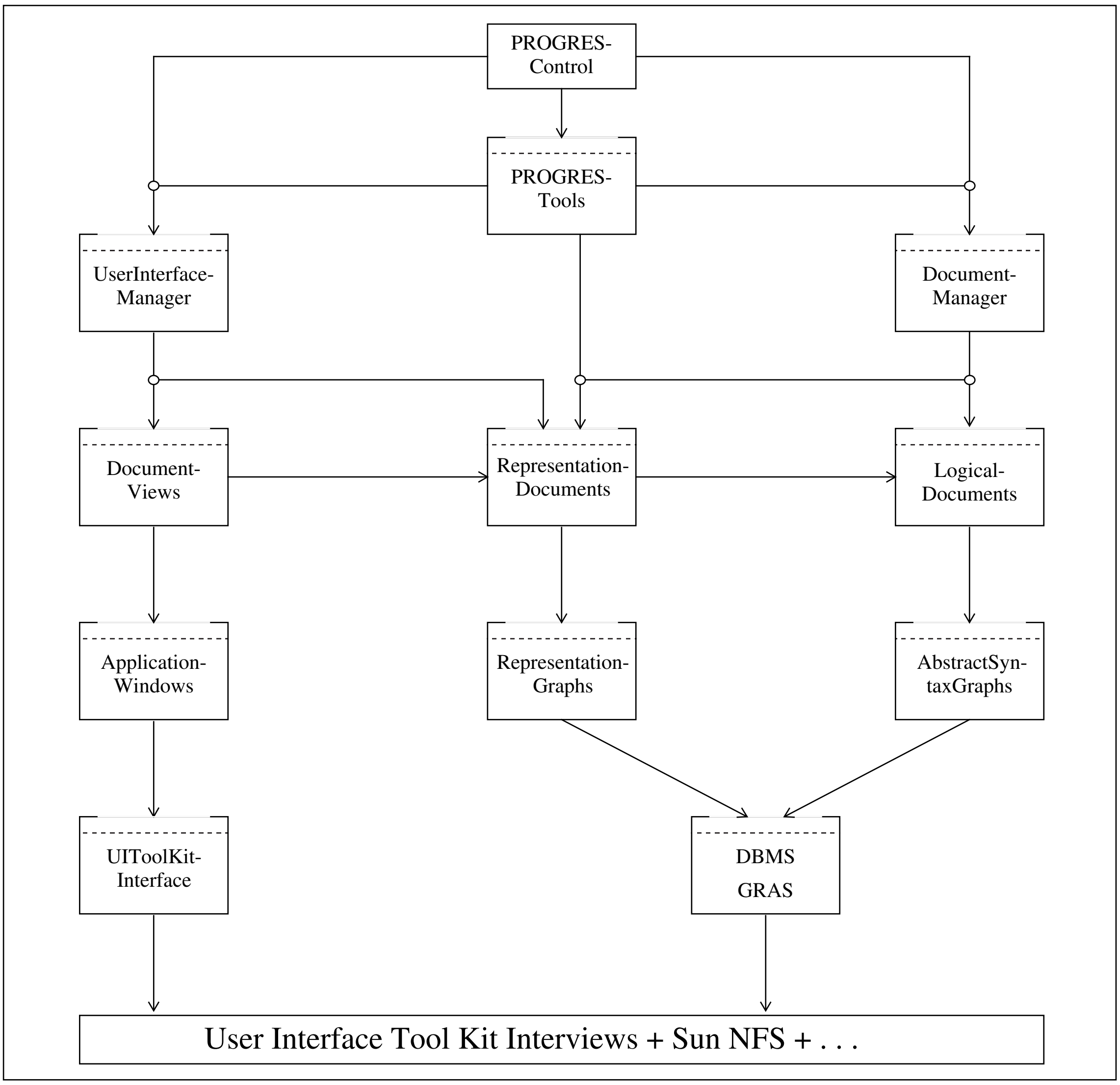


Fig. 5: Overall PROGRES environment architecture.

(1) The root of the system is a small module with name ProgresControl that coordinates the activities of all available tools.

(2) The next layer contains implementations of all PROGRES tools which are responsible for translating (abstract) user interface events into modifications of internal data structures or vice versa.

(3) On the third layer from top to bottom we find two subsystems that manage the flow of information between all related data structures in the environment and are responsible for keeping window contents consistent with their underlying specification documents.

(4) The fourth layer realizes three different categories of abstract data types, which are (from right to left): LogicalDocuments store all abstract syntax, type checking, and runtime information about specification documents, RepresentationDocuments handle all additional information about their concrete syntax, layout, visibility etc., and DocumentViews display specifications at the user interface.

(5) The next layer offers resources for implementing DocumentViews as overlapping Windows, RepresentationDocuments as a special kind of RepresentationGraphs, and finally LogicalDocuments as AbstractSyntaxGraphs.

(6) The last layer on top of the operating system's resources (including the user interface tool kit InterViews[6]) consists of a small abstraction layer on the user interface side, and a huge subsystem implementing our own distributed DBMS GRAS.

Returning to our list of requirements, we will see that most of them influenced directly the PROGRES system's design in figure 5. *Uniformity* of tools is for instance guaranteed by the fact that all tools have to play well-defined roles within the presented architecture framework and use the same basic components.

The most important basic component is the (sub-)system GRAS mentioned above, a *non-standard database management system* with a graph-oriented data model and a multiple client/server architecture. It supports efficient manipulation of persistent graph structures, incremental evaluation of derived graph properties, nested transactions, undo&redo of arbitrarily long sequences of already committed transactions, recovery from system crashes, and has various options of how to control access of multiple clients to their data structures. In this way, *persistency* of tool activities including undo&redo and recovery comes for free. And *integration* of tools is mainly achieved by storing all data within GRAS and by using two management subsystems for *incremental* maintenance of all data dependencies.

The requirement *structure-oriented* manifests itself in the existence of the subsystem AbstractSyntaxGraphs, whereas the properties *command-driven* and *interactive* are not directly visible on the coarse system design level but are taken into account in the implementation of each subsystem. Finally, the fulfillment of the requirement *multi-view-oriented* is reflected in the arrangement of the overall system architecture into three columns dealing with bitmaps and windows, textual and graphical representation elements, and information about a specification document's abstract syntax and semantics (from left to right). The following section discusses the relationships between these three different categories of data in more detail.

---

6) Developed at Stanford University and available via anonymous ftp from interviews.stanford.edu.

4. The PROGRES Editor and Analyzer

The PROGRES editor and analyzer are those tools of our environment that assist its users when 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 avoided and the environment's user is allowed to switch freely between editing, analyzing, and debugging activities back and forth[7]. 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), manual rearrangement of text and graphic elements, and finally browsing and searching activities. Figure 6 contains a rough sketch of all involved subtools and their data structures.

Syntax-directed editing works as follows: The active graphic window (with an attached menu area) belongs to an active representation document that contains a pointer to the currently selected representation element. This element in turn has a link to the corresponding increment in the logical document[8]. Being aware of the PROGRES language's context-free syntax, the editor offers only those commands in the active window's menu area which pre-

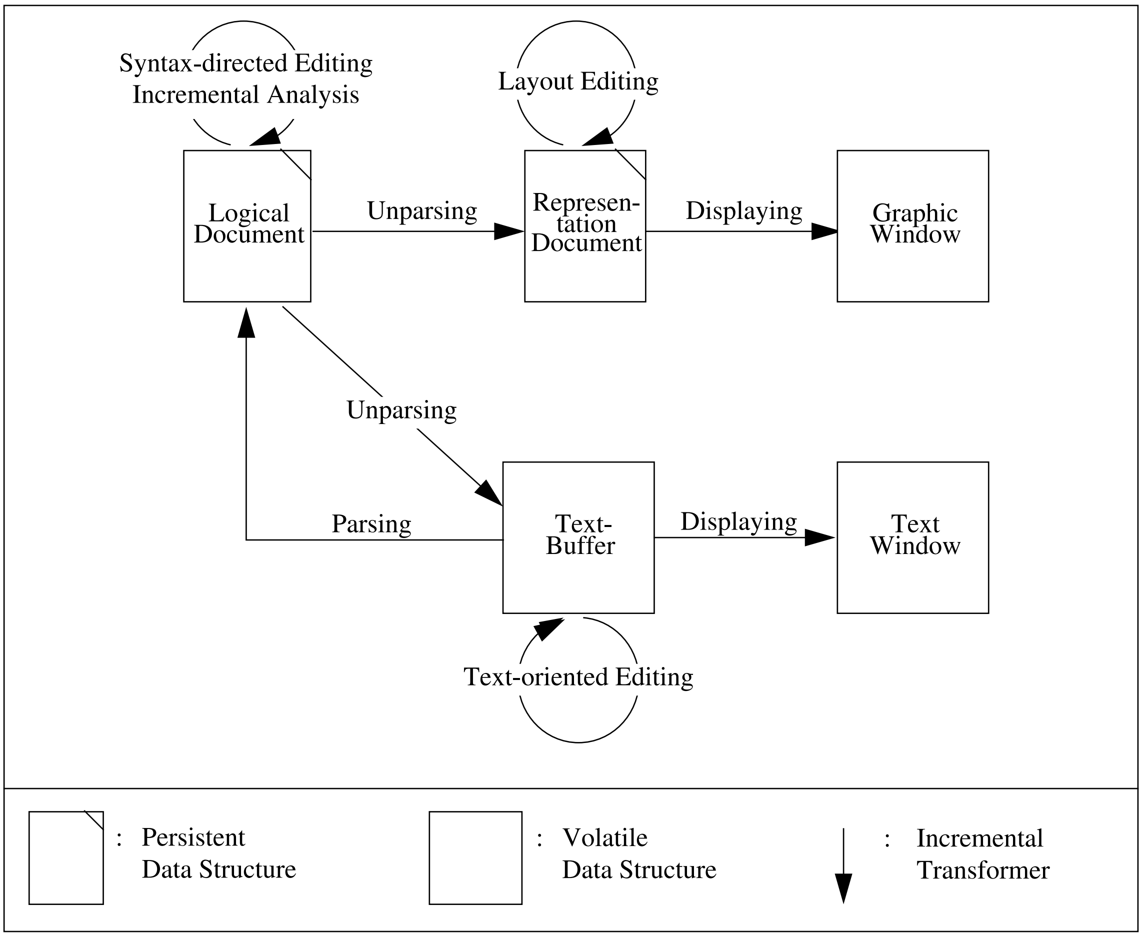


Fig. 6: Editor data structures and transformation processes.

7) With the exception of graph scheme modifications which are disallowed while the interpreter is running.

8) A logical document has e an arbitrary number of different representation documents and a representation document may be displayed in an arbitrary number of graphic windows.

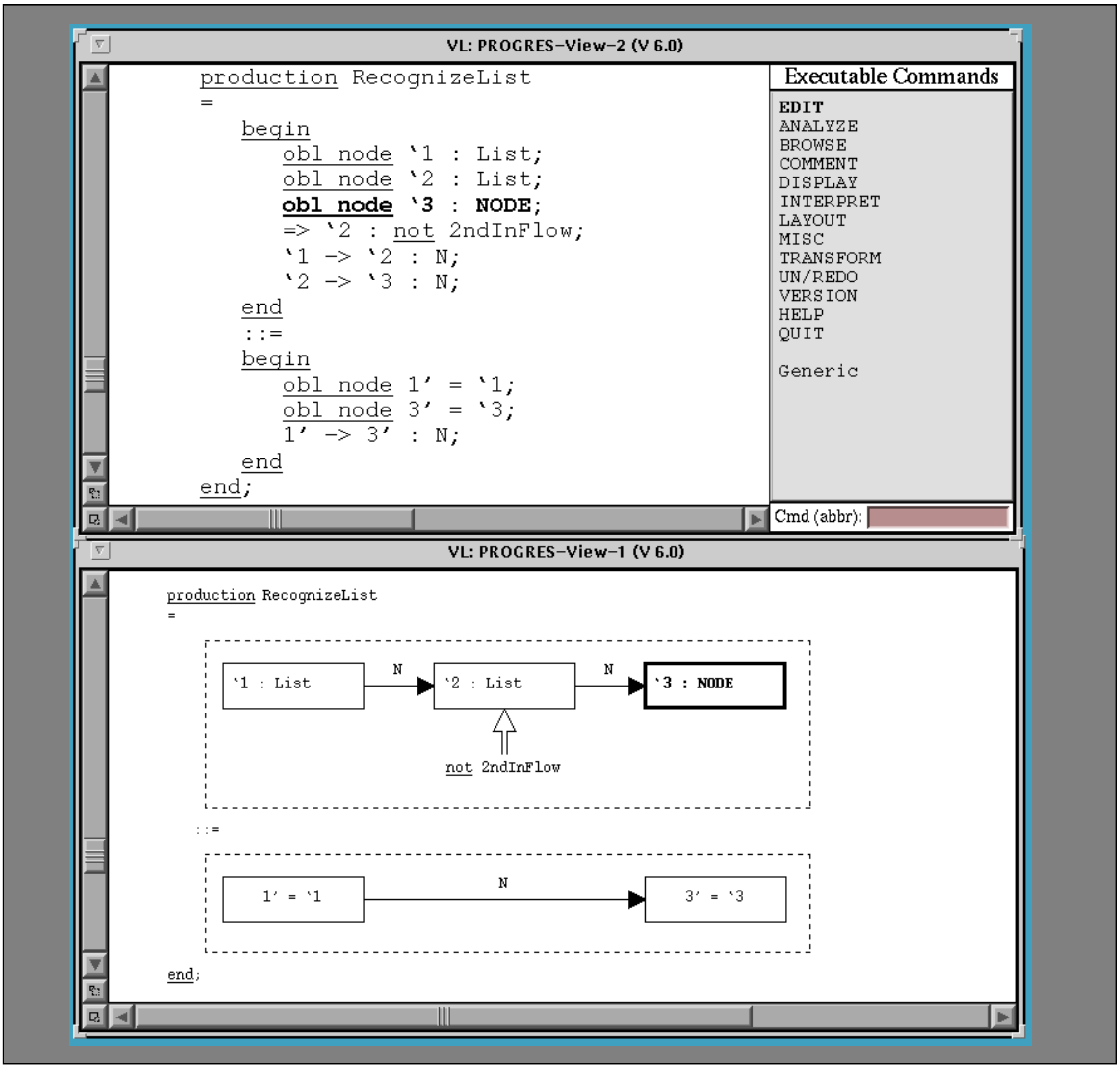


Fig. 7: Diagrammatic and textual representation of a production.

serve context-free correctness. The execution of a selected command starts with the modification of the logical document's abstract syntax tree skeleton and markingof all affected document parts for deferred unparsing and analysis activities.

When all necessary updates of the logical document are completed, then a table-driven unparser propagates all modifications to all related representation documents. This process is divided into the following steps: The first step is to remove all representation elements with dangling references to no longer existing logical increments, to update all representations of all modified increments, and to insert new representation elements for all new logical increments. The next step is to recompute the layout of all affected representation elements[9] and to compute a list of smallest enclosing areas of modified representation document parts. Finally, when all document modifications are processed and a consistent state is required at the user interface, all affected and visible representation areas are redisplayed. Now, a new current increment may be selected by pointing to a location in a window, where the desired increment is visible.

9) The computed layout may be modified afterwards.

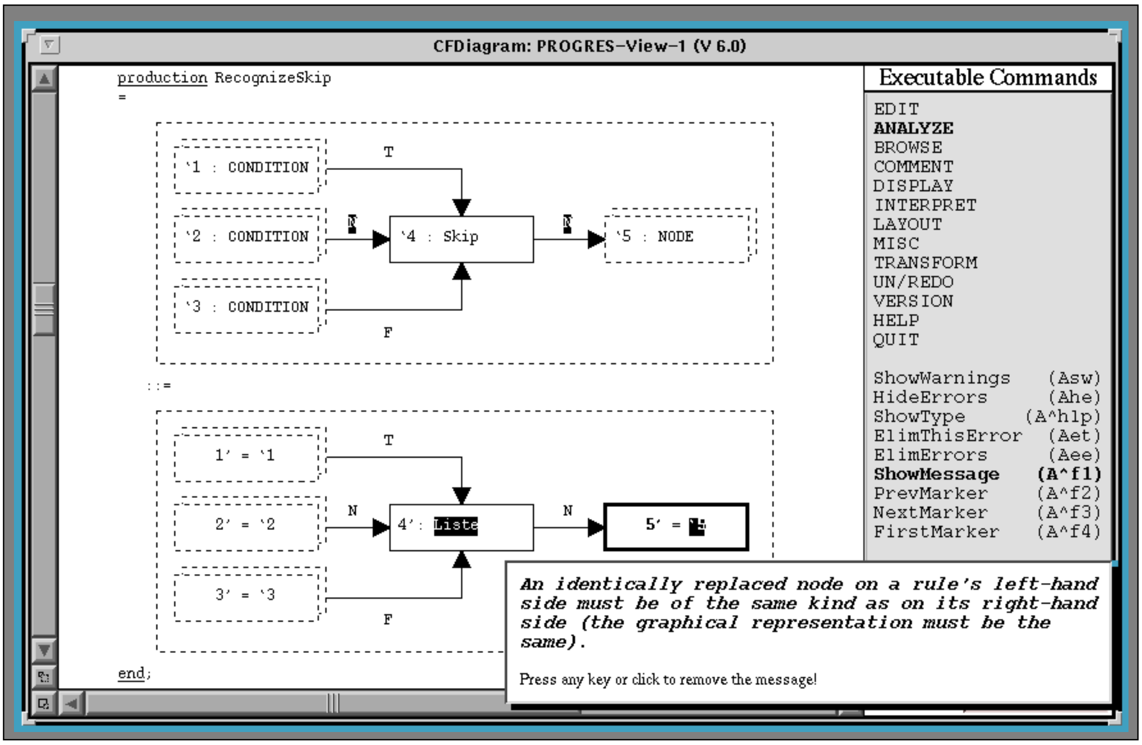


Fig. 8: Production with four static semantics errors.

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 language'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 language constructs, even for those that have usually a graphical representation[10] as the production RecognizeList of figure 7.

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 6).When text processing is finished, a multiple-entry LALR-parser[11] 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 too guarantee reasonable response times. Significant changes at a root class of a graph scheme's inheritance hierarchy requires for instance reanalysis of a very large fraction of the corresponding specification document.

10) All graphical language constructs possess an equivalent textual substitute.

11) "Multiple-entry" means that the parser is not only able to handle complete specification texts derived from the language's axiom, but also any text fragments that are derived from other nonterminals.

When being active, incremental analysis starts with a set of all modified/labeled abstract syntax subtrees in a logical document and determines all (applied) identifiers the bindings of which must be recomputed. When all identifier bindings are reestablished, an *acyclic dependency graph* is constructed that contains all previously marked document subtrees, all identifiers with new bindings, and all those subtrees within the document the static semantics of which depend on the static semantics of subtrees that are already part of the dependency graph. Analysis itself processes one subtree after the other in the given dependency order, stores derived type-checking information in the form of additional edges and attributes[12], and labels all erroneous specification parts. In this way, our analyzer recognizes more than 300 different types of errors - defined in [47] - and it offers explanations in the form of error messages on demand.

The production RecognizeSkip of figure 8 contains four examples of typical programming errors (increments with black background):

(1) The graph scheme of figure 2 requires that N(ext)-edges have BLOCK nodes as their sources. Therefore, N(ext) is not a proper label for an edge starting at the CONDITION node ‘2 in the production’s left-hand side (CONDITION is not a subclass of BLOCK).

(2) The second N(ext)-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.

(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 8. 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 here provide its users with substantial assistance for creating errorfree specifications. The hybrid syntax-directed and 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.

The complete editor code is not hand-crafted but generated with our own *IPSEN meta environment* (cf. [35, 47]). It takes a language’s EBNF plus additional text and graphic unparsing annotations as input and generates all language-specific editor modules that are necessary for context-free editing and parsing. The language-specific parts of the analyzer are hand-written but research is under way to bootstrap PROGRES, i.e. to specify the language’s static semantic’s using the language itself and to use the PROGRES compiler of the next section to translate the resulting specification into efficiently executable code of a conventional imperative programming language.

12) Remember that logical documents are internally realized as attributed graphs.

5. The PROGRES Interpreter and Compiler

The user of the PROGRES environment has two alternatives how to animate a given specification. The first one is based on direct interpretation and it is mainly used for debugging purposes, when intertwining of editing, analyzing, and execution activities is advantageous. The second one is to translate a specification into equivalent Modula-2 or C code and to use the generated source code together with the graph DBMS GRAS as (a rapid prototype of) an abstract data type implementation in a larger program. This program may contain additional components for the realization of a user interface, interprocess communication etc.[13] In both cases, the execution process is divided into two phases:

(1) The initialization phase creates an empty host graph in GRAS, processes all graph scheme declarations and translates them into an internal format with a static derived data dependency graph as its main component. GRAS needs this dependency graph to maintain all derived data (on demand) in a consistent state.

(2) Afterwards, arbitrary productions or transactions may be executed step by step, and they may even be modified and recompiled while the execution process is running.

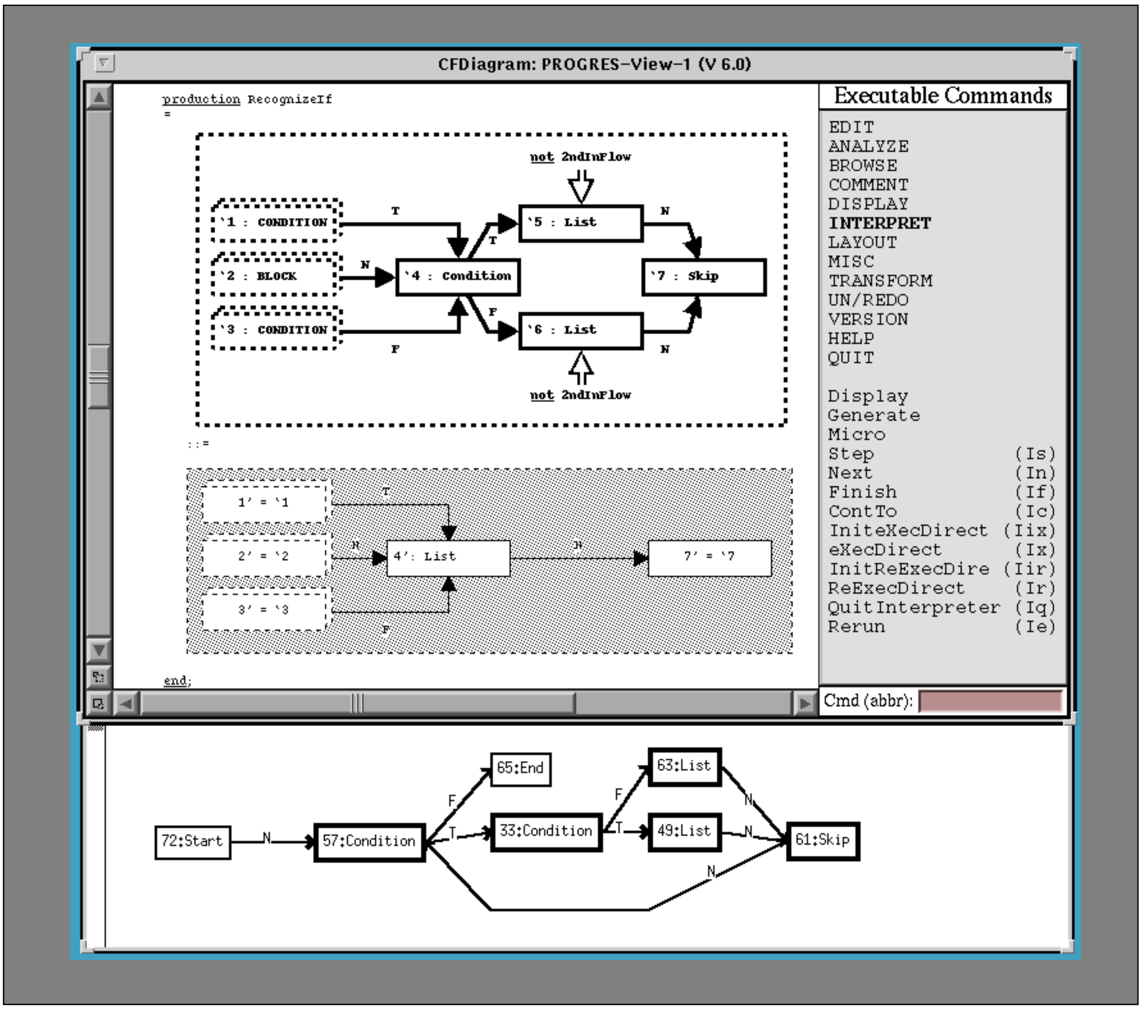


Fig. 9: Applying a production to a given host graph.

13) PROGRES supports specification of graph-like data structures and operations on these data structures but not the design of user interfaces etc.

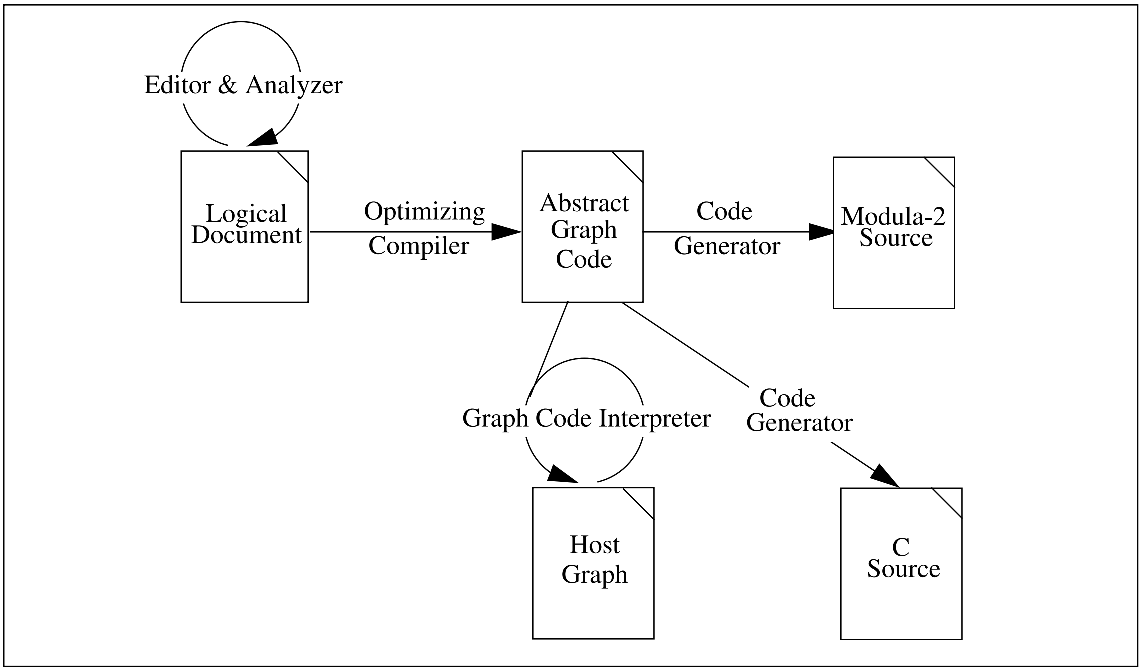


Fig. 10: Execution data structures and transformation processes.

Figure 9 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 diagram), and the production's right-hand side is the current interpreter increment. The environment's user has now the choice either to execute the replacement of the recognized pattern by a copy of the right-hand side step by step or to finish the replacement in one step. In both cases, the selected Condition and List nodes will be deleted with all attached edges, the selected skip node will be preserved (identical replacement), and finally a new List node will be created and embedded into the remaining control flow diagram by drawing a N(ext) edge from node 57 to the new node and another N(ext) edge from the new node to node 61. Afterwards, a new production will be selected automatically or may be determined by hand that recognizes another pattern within the resulting control flow diagram.

The diagram in figure 10 gives an overview of the main components and data structures which are involved in the above mentioned activities. It reveals that the editor and analyzer of section 4 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 (on demand) into intermediate code for an abstract graph rewriting machine. The compiler is the most important component of the whole execution machinery and determines its efficiency to a great extent. Especially in the case of productions, where we have choose between a large number of different intermediate code sequences, which perform the required subgraph matching but differ with respect to runtime efficiency considerably[14] (for further details cf. [55]).

14) Varying from O(n) to O(n*m^n) where n is the number of nodes in a production's left-hand side and m is the number of nodes in a given host graph.

The produced abstract graph code may be executed directly or serves as input for two compiler backends which produce equivalent Modula-2 or C code. The abstract graph rewriting machine combines the functionality of a conventional stack machine à la P-code [40] with backtracking capabilities à la Warren Abstract Machine [51]. Furthermore, its main component offers facilities for manipulating persistent graph structures stored in the database system GRAS.

The translation of abstract graph code into readable Modula2 or C source code is rather straightforward with the exception of backtracking, which requires reversing a 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 [54]. It follows the lines of a proposal presented in [29] how to embed backtracking constructs into conventional C programs.

Compiled PROGRES specifications are about a factor of 10 faster than interpreted abstract graph code. The reasons for such an unusually low interpretation overhead are twofold:

- Parsing, analyzing, and intermediate code generation is performed once and for all, and interpretation is done on the level of optimized abstract graph code only.
- The main speedup is gained when processing conventional language elements like function calls, expression evaluation etc. In all other cases, the overhead for decoding abstract graph code is negligible compared with the complexity of the evoked GRAS database system operations.

To summarize, the PROGRES environment tries to combine the flexibility of interpretative approaches with the efficiency and safeness of compilative approaches, and it processes user inputs in four interlacing phases:

(1) Editing or incremental parsing modifies a specification's underlying abstract syntax tree.

(2) Incremental analysis of affected subtrees adds inferred type information to a given abstract syntax tree on demand.

(3) Lazy recompilation invalidates inconsistent fragments of abstract graph code and updates them on demand.

(4) Compiler backends may be activated to translate (pieces of) abstract graph code into source code of imperative programming languages.

6. Related Work

When comparing the functionality of programming languages and their environments, it is often difficult to distinguish between properties of a language and properties of its accompanying tools. Therefore, we will not try to discuss related work about visual languages and about visual environments separate from each other, and we will even focus our main interest onto those approaches where both a *language and an environment* exists and where both earn the attribute "*visual*".

Another problem for a comparison of PROGRES with other approaches stems from the fact that PROGRES is a *multi-paradigm language* which shares at least some properties with a large fraction of all existing (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 [20, 24]. Furthermore, we decided to disregard all those approaches which aim at the recognition (parsing) of diagram languages only and have a much more limited application domain [21, 27, 31].

Using these criteria, a surveyable number of visual languages remain for inspection. Compared with PROGRES their deficiencies come from the following sources: in many cases these languages focus on *manipulation of data structures only* and data definition sublanguages are not provided [3, 22, 28]. And even systems like VAMPIRE with its class hierarchies and icon rewriting rules [30] and PAGG with its node type definitions and graph rewriting rules [17] 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 typeless or weakly typed programming language.

With respect to its graph scheme definition capabilities, PROGRES is more similar to so-called visual database query or programming languages [1, 2, 9, 10]. But these languages have less expressive pattern matching and replacing constructs (no object set patterns etc.), and some of them are even not computational complete (no recursion etc.). Furthermore, neither the above mentioned languages nor, to the best of our knowledge, any other diagrammatic rule-oriented language offers *nondeterministically* working control structures together with the ability to *backtrack* out of dead-ends of locally failing rewriting processes.

7. Conclusion

This paper contains a brief presentation of the *visual graph grammar based language* PROGRES and a more detailed discussion of its accompanying programming environment. The PROGRES language has a well-defined syntax and semantics and provides its users with declarative as well as procedural and diagrammatic as well as textual constructs for the definition of graph schemes, derived graph properties, graph rewrite rules with elaborate pattern matching constructs, and complex graph transformation processes. The PROGRES environment offers means to create, analyze, compile, and debug 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 (statically typed) languages and their environments.

A first version of the PROGRES environment with about 400.000 lines of code is already available via anonymous ftp[15] for Sun workstations. It supports editing and execution of all language elements but is still subject to improvements with respect to runtime efficiency. Furthermore, we have started to develop new concepts and tools for "*programming in the large*", i.e. for decomposition of large specifications into separate, reusable, and encapsulated modules.

---

15) From ftp.informatik.rwth-aachen.de [137.226.112.172] in /pub/unix/PROGRES

References

[1] Andries M., Engels G.: *Syntax and Semantics of Hybrid Database Languages*, in: [16]

[2] Angelaccio M., Catarci T., Santucci G.: *QBD*: A Graphical Query Language with Recursion*, in: IEEE Transactions on Software Engineering, vol. 16, no. 10, Los Alamitos: I IEEE Computer Society Press (1990), 1150-1163

[3] Arefi F., Hughs Ch.E., Workman D.A.: *Automatically Generating Visual Syntax-Directed Editors*, in: CACM, vol. 33, no. 3, New York: acm Press (1990), 349-360

[4] Borras P. et al.: *CENTAUR: The System*, in: ACM Software Engineering Notes, vol. 13, no. 5, New York: acm Press (1988), 14-24

[5] Bunke H.: *Sequentielle und parallele programmierte Graphgrammatiken*, Technical Report IMMD-7-7, Universität Erlangen, Germany (1974)

[6] Cattell R.G.G. (ed.): *The Object Database Standard: ODMG-93*, San Mateo: Morgan Kaufmann Publ.

[7] Chang SH.-K., Ichiakwa T., Ligomenides P.A. (eds.): *Visual Languages*, New York: Plenum Press

[8] 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)

[9] 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

[10] Catarci T., Santucci G.: *Query By Diagram: A Graphic Query System*, in: *Entity-Relationship Approach*, Amsterdam: Elsevier Science Publ. B.V. (1989), 291-308

[11] Engels G., Lewerentz C., Schäfer W.: *Graph Grammar Engineering - A Software Specification Method*, in: [14],186-201

[12] 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

[13] Ehrig H., Nagl M., Rozenberg G. (eds.): *Proc. 2nd Int. Workshop on Graph Grammars and Their Applications to Computer Science*, LNCS 153, Berlin: Springer Verlag (1982)

[14] Ehrig H., Nagl M., Rozenberg G., Rosenfeld A. (eds.): *Proc. 3rd Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 291, Berlin: Springer Verlag (1986)

[15] 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),

[16] Ehrig H., Schneider H.-J.: *Proc. Dagstuhl-Seminar 9301 on Graph Transformations in Computer Science*, LNCS 776, Berlin: Springer Verlag (1994)

[17] Göttler H., Günther J., Nieskens G.: *Use Graph Grammars to Design CAD-Systems*, in [15], 396-410

[18] Glinert E.P.: *Visual Programming Environments: Paradigms and Systems*, Los Alamitos: IEEE Computer Society Press (1990)

[19] Glinert E.P.: *Visual Programming Environments: Applications and Issues*, Los Alamitos: IEEE Computer Society Press (1990)

[20] Glauert J.R., Kennaway J.R., Sleep M.R.: *Dactl: An Experimental Graph Rewriting Language*, in: [15], 378-395

[21] Golin E.J., Reiss St.P.: *The Specification of Visual Language Syntax*, in: [56]*, 105-110*

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

[23] Kilian M.F.: *A Note on Type Composition and Reusability*, in: OOPS Messenger, vol. 2, no. 2, New York: acm Press (1991), 24-32

[24] 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

[25] Kreowski H.J.: *Grace, a graph centered environment*, invited talk of COMPUGRAPH Workshop, Nordwijkerhout, 3-7 October, Netherlands (1993)

[26] Kiesel N., Schürr A., Westfechtel B.: *Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications*, in: CASE '93 Proc. 6th Int Workshop on Computer-Aided Software Engineering, Los Alamitos: IEEE Computer Society Press (1993), 272-286

[27] Lakin F.: *Spatial Parsing for Visual Languages*, in: [7], 35-86

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

[29] Liu Y.: *btc: A Backtracking Procedural Language*, Technical Report 203, University of Queensland, Key Center for Software Technology, Australia (1991)

[30] McIntyre D.W., Glinert E.P.: *Visual Tools for Generating Iconic Programming Environments*, in: [57], 162-168

[31] Meyer B.: *Pictures Depicting Pictures: On The Specification of Visual Languages by Visual Grammars*, in: [57], 41-47

[32] Nagl M.: *GRAPL - A Programming Language for Handling Dynamic Problems on Graphs,* in: [39], 25-45

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

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

[35] Nagl M.: *A Characterization of the IPSEN-Project*, in: Proc. Int. Conf. on System Development Environments & Factories, Pittman (1990) 141-150

[36] Nagl M., Schürr A.: *A Specification Environment for Graph Grammars*, in: [15], 599-609

[37] Newbery Paulisch F.: *The Design of an Extendible Graph Editor*, LNCS 704, Berlin: Springer Verlag

[38] Pape U.: *GRAMAS - A Graph Manipulation System*, in: [39], 47-63

[39] Pape U. (ed.): *Discrete Structures and Algorithms*, Proc. WG '79, 5th Int. Workshop on Graph-Theoretic Concepts in Computer Science, München: Hanser Verlag (1979)

[40] Pemberton St., Daniels M.: *Pascal Implementation: The P4 Compiler*, Ellis Horwood Ltd. (1982)

[41] Pfaltz J., Rosenfeld A.: *Web Grammars*, Proc. Int. Joint Conf. on Artificial Intelligence, S. 609-619 (1969)

[42] Reiss St.P.: *Working in the Garden Environment for Conceptual Programming*, IEEE Software, vol. 4, no. 6, Los Alamitos: IEEE Computer Society Press (1987), 16-27

[43] Rekers J.: *On the Use of Graph Grammars for Defining the Syntax of Graphical Languages*, submitted for publication from Leiden University, Netherlands (1994),

[44] Reps Th.W., Teitelbaum T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Berlin: Springer Verlag (1989)

[45] Schneider H.J.: *Chomsky-Systeme für partielle Ordnungen*, Arbeitsbericht IMMD-3-3, Universität Erlangen, Germany (1970)

[46] Schürr A.: *PROGRES: A VHL-Language Based on Graph Grammars*, in: [15], 641-659

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

[48] Schürr A.: *Logic Based Structure Rewriting Systems*, in: [16], 341-357

[49] Schürr A., Zündorf A.: *Nondeterministic Control Structures for Graph Rewriting Systems*, in: Proc. WG'91 Workshop in Graphtheoretic Concepts in Computer Science, LNCS 570, Berlin: Springer Verlag (1992), 48-62

[50] Schwartz J., Westfechtel B.: *Integrated Data Managment in a Heterogeneous CIM Environment*, in: Proc. CompEuro ’93, Los Alamitos: IEEE Computer Society Press, 272-286

[51] Warren D.H.D.: *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, Menlo Park

[52] Westfechtel B.: *Revision Control in an Integrated Sofware Development Environment*, in: Proc. 2nd Int. Workshop on Software Configuration Management, ACM Software Engineering Notes, vol. 17, no. 7, New York: acm Press (1989), 96-105

[53] Westfechtel B.: *Structure-Oriented Merging of Revisions of Software Documents*, in: Proc. 3rd Int. Workshop on Software Configuration Management, New York: acm Press (1991), 68-80

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

[55] Zündorf A.: *A Heuristic Solution for the (Sub-)Graph Isomorphism Problem in Executing PROGRES*, Technical Report AIB 93-5, RWTH Aachen

[56] Proc. of the 1989 IEEE Workshop on Visual Languages, Los Alamitos: IEEE Computer Society Press (1989)

[57] Proc. of the 1992 IEEE Workshop on Visual Languages, Los Alamitos: IEEE Computer Society Press (1992)