Graph Grammar Engineering with PROGRES

Andy Schürr[1], Andreas J. Winter[1], and Albert Zündorf[2]

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

2. AG Softwaretechnik, Fachbereich 17, GH-Paderborn, Warburger Str. 10, D-33098 Paderborn, Germany; e-mail: zuendorf@uni-paderborn.de[3]

Abstract: Graph-like data structures and rule-based systems play an important role within many branches of computer science. Nevertheless, their symbiosis in the form of graph rewriting systems or graph grammars are not yet popular among software engineers. This is a consequence of the fact that graph grammar tools were not available until recently and of the lack of knowledge about how to use graph grammars for software development purposes. "Graph grammar engineering" is a first attempt to establish a new graph and rule centered methodology for the development of information system components. Having its roots in the late 80's it gradually evolved from a "paper and pencil" specification formalism to a tool-assisted specification and rapid prototyping approach.

1 Introduction

Graphs or graph-like data structures play an important role within many branches of computer science. Entity-Relationship diagrams are commonly used to design database systems, control and data flow graphs are important for the realization of optimizing compilers, and conceptual graphs are popular among Artificial Intelligence researchers.

Furthermore, *rule-based languages* and systems are getting more and more important for the specification and implementation of complex information systems. As a matter of fact, both graph-like data structures and rule-based rewriting processes are essential parts of a growing number of software systems. Nevertheless, their "natural" combination in the form of graph rewriting systems or graph grammars are almost unknown among computer scientists. One reason for this shortfall of 25 years of graph grammar research [GG78, GG82, GG86, GG95] is its concentration on producing theoretical results. Therefore, publicly available graph grammar based tools or languages did not exist until recently. Furthermore, there was a considerable lack of knowledge about how to use graph rewriting systems for specification purposes (cf. [BFG95] for a summary of graph grammar research from a practical user's point of view).

---

3. Supported by: Deutsche Forschungs-Gemeinschaft

Very first results of more application-oriented research activities are represented by the systems

- GraphEd [Him90] with its main emphasis on graph drawing (and recently graph parsing),
- PAGG [GGN90], the oldest running graph grammar programming environment, which aimed at the application area of CAD systems, and
- AGG [LöBe93], a learner's environment for pushout based graph rewrite rules.

All these systems are far from being usable as real specification or very high level programming environments. Nowadays, there exists only one notable exception from this state of graph grammar research, the language *PROGRES* and its *integrated programming environment*. Both the language and its tools are the result of many years of experience with using graphs and graph rewriting systems within the research project *IPSEN*. Within this project, graph (grammar) technology was and is quite successfully used for developing Integrated Project Support Environments. The papers [ELS86] and especially [ELNSS92] document the growth of knowledge on how to model internal data structures of software engineering environments as graphs and how to specify operations on them by means of graph rewriting rules.

Based on these experiences, we have developed

- a visual and operational *specification language*,
- an *integrated set of tools* for editing, analyzing, and executing specifications in this language, and
- an accompanying *methodology* concerning the use of the language and its tools.

This methodology (and its predecessors) are termed *graph grammar engineering*, inspired by the long-term goal to develop a new graph-oriented software engineering approach. It is no longer restricted to the application area of software engineering environments and relies on well-known *object-oriented modeling techniques* for designing graph data structures and graph schemata à la [Boo94, RBPEL91], but it has a rather different approach for modelling operations on these data structures. Object-oriented techniques concentrate on the design of object(classe)s, on the encapsulation of internal object data, and on the methods used to access this object data. On the other hand, graph grammar engineering concentrates on complex object structures, on the organization of objects, and on the modelling of relationships between objects. Therefore, graph grammar engineering offers specialized query and update operations using complex graph pattern matching and replacement rules.

This paper presents the current state of graph grammar engineering as a process that starts with an informal requirements definition and ends with an efficiently working realization. It discusses each step of the new software development strategy by means of a running example, the realization of a simple medical diagnosis (expert) system. It is out of scope for this paper to give a detailed description of the language PROGRES and its implementation. For more information about these subjects the reader is referred to [Sch91a, Sch91b, Zü95].

2 Graph Grammar Engineering Overview

Our new software development methodology is not intended to be used for any kind of software. On the contrary, it is aimed at a special, yet very important class of software components, so-called *information (sub-)systems*, which have the following characteristics:

- They deal with complex data structures with lots of cross-references and derived information.
- They perform complex pattern matching and transformation tasks on their data structure, including backtracking implementations of inference processes.
- User interfaces and any kind of input/output tasks are treated within separate subsystems (so that different technologies may be used for their development).

A software (sub-)system with these properties may be developed in the following manner using our graph grammar engineering methodology:

1. **Gathering informal requirements**: Let us assume that an informal requirements definition is already available. It may contain sample cases of processed data (modelled as directed graphs) and informal descriptions of its most important and complex operations (including subsequent data snapshots of their intended effects).
2. **Designing the graph schema:** The next step is to analyze the structure of given sample graphs and to identify *types* of objects (nodes), their *common properties*, dependencies, and *relations* between them. This will lead to some kind of multiple inheritance node class hierarchy including the definition of derived attributes and relationships. This step should be done carefully since it is a very important but difficult process within the development of information systems. This process is supported by the PROGRES development environment by checking types and consistency.
3. **Specifying graph transformations:** Based on the graph schema, consistency preserving *operations* can be specified that represent all permissible graph queries and transformations. This can be done easily using the powerful high level notation of graph rewriting rules. Within the PROGRES environment, static analysis ensures the schema consistency of operations and, thereby, detects almost all typing errors and many more subtle inconsistencies. Considerations about efficiency should be disregarded at this point. Rather, the designer should derive benefit from declarative and rule-based character of the language.
4. **Validating the specification's correctness:** Having finished the specification of graph schema and operations, it can be validated with the *PROGRES interpreter* which is available as an integrated part of the development environment. The PROGRES user may even switch back and forth between editing and execution activities (within certain limits). A graph browser with automatic graph layout helps the user to keep track of the effects of the specified graph rewriting operations.
5. **Improving efficiency:** As soon as the specification is validated, some aspects of *efficiency improvement* can be considered. With the help of additional keywords, the user can introduce index structures for attributes and complex traversal operations

that allow associative access to nodes. Or, one can guide the graph pattern matching process by providing some nodes of the searched subgraph as parameters to the rewriting rule. Also, enhancing the data structure itself by adding additional references can improve the pattern matching process significantly.

6. **Generating a prototype:** Eventually, the specification is "usable". By means of generators and the PROGRES compiler, the designer is now able to *create a prototype* of the modelled system. This prototype is based on a standard architecture for interactive systems, a simple, freely available toolkit for generating graphical user interfaces, and a graph browser. The generated prototype consists of several display components and menus for calling interface operations of the modelled system. For operations that need parameters, input windows are generated. In this way, the user can interactively work with the system and observe its current state with the graph browser.
7. **Deriving an implementation:** If the generated prototype is satisfying, the *final implementation* may be produced. For this purpose, the PROGRES compiler generates C or Modula-2 code which is semantically equivalent to the specification given in PROGRES. The generated code can be integrated into the frame architecture of the system. The generated source code has the important quality of being human readable, e. g. the names of identifiers in the specification are maintained, the original specification is textually included in the source code as comments and a pretty printer takes care of correct indentation of the control structures. This facilitates adding new functionality to the generated code and opens the possibility to tune and optimize critical parts of the implementation.

In the following, we will present a more detailed view of graph grammar engineering by applying it to the development of a medical diagnosis system for which we want to generate a prototype and therefore perform the steps outlined above. This allows us to present some highlights of the PROGRES language and to sketch how the PROGRES environment supports the development process.

3 Gathering Informal Requirements

The example application is an "simple" expert system that helps a doctor to find the correct diagnosis for the symptoms described by a patient. This puts him in the situation to determine the suitable treatment. The system is supposed to have different points of view. First, it keeps track of all treatments by different doctors and manages the billing due to the resulting costs of treatments and medicine. On the other hand, it is a knowledge-based system that suggests the application of a certain treatment or leads to further examinations which eventually reveal the actual disease the patient suffers from. The system shall also notify the doctor of possible contraindications to certain treatments like e. g. allergies to any medicine. In order to be able to perform this task, we have to model the dependencies between several symptoms, the diseases that cause them, the examinations that reveal them, and the treatments that ease them.

According to our graph grammar engineering approach, we begin the system development process with the graphical design of some data snapshots. Figure 1 shows such an example situation. The upper half shows the administrative portion, i. e. which patient attends which practice and the doctors working in it. For every patient the list of his treatments is known and also which doctor performed them.

The lower part of the figure explains the medical knowledge already acquired, e. g. simple symptoms as sore throat and cough, examinations as taking temperature, diseases as tonsillitis and diphtheria, and treatments as antibiotics or antipyretics. There are dependencies between symptoms, examinations, and diseases that are represented by *pro*- and *contra*-edges. Note that at some points we cannot clearly distinguish between symptoms and diseases, but in most cases we can decide whether a symptom is an indication for or against some disease. Similarly, examinations give a clue for the presence or absence of certain diseases. The fact that a disease can be cured by a medicine is expressed by a *needs*-edge,

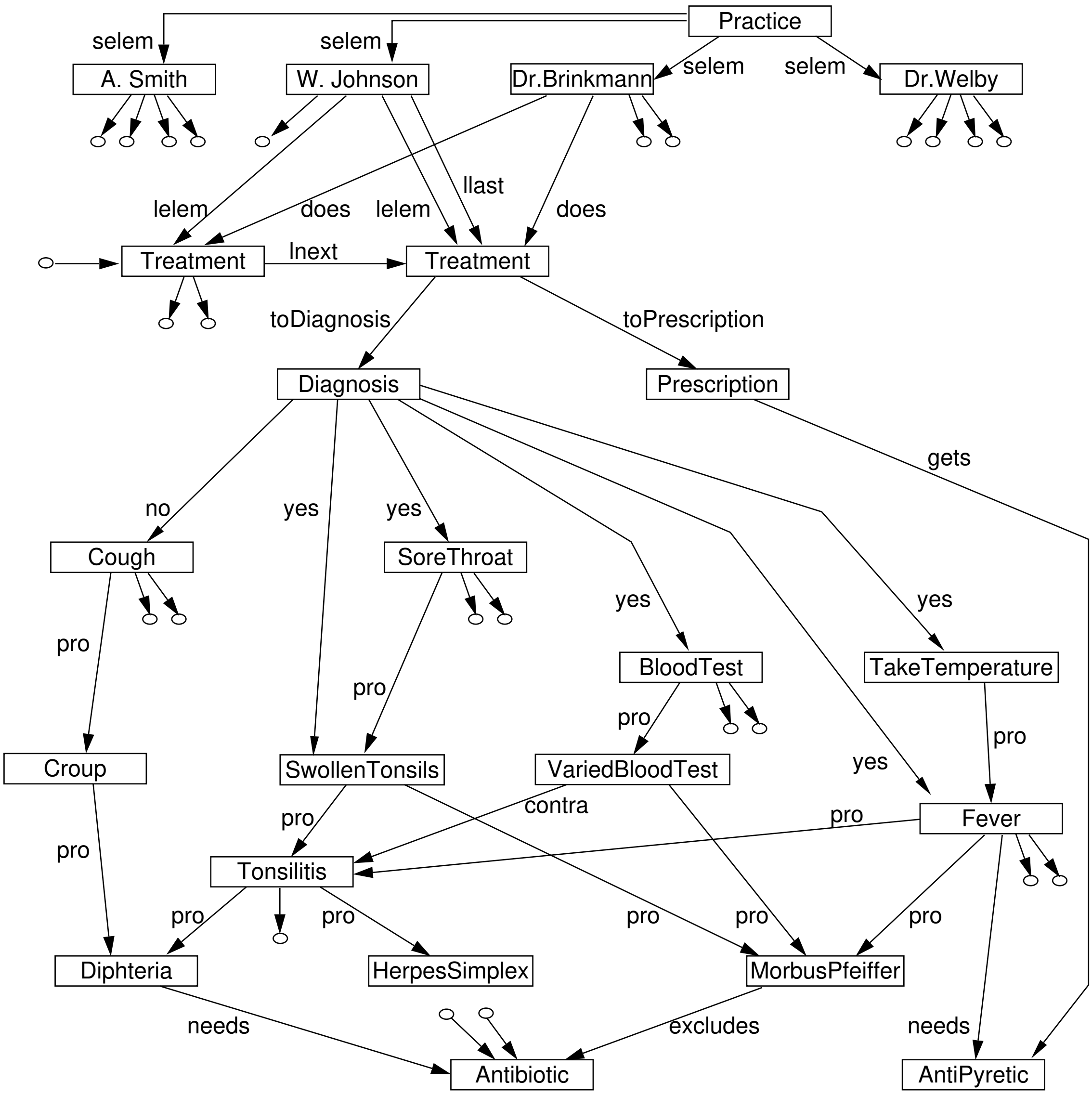


Figure 1: Example scenario for a medical diagnosis system

whereas an *excludes*-edge represents the intolerance of a disease with some medicine. The results of examinations are given by *yes*- and *no*-edges that connect diagnoses to the corresponding nodes in the knowledge graph. The treatment prescribed by the doctor is represented by *gets*-edges from prescriptions to medicine and other therapy methods, respectively.

The model needs to be extended with information about the costs of examinations or therapies. This can be done by means of suitable attributes so that the overall costs of a treatment can be determined by the sum of the parts it consists of. In the same way, the turnover of a certain doctor or practice can be calculated or the share of a special treatment to the overall costs.

We now take a closer look at the operations that our diagnosis system offers. First, there are operations to establish a doctor's practice and to acquire new patients, then we need operations to record the results of examinations and tests. The operation in Figure 2 shows the idea of starting a new treatment for a patient. The meaning of this rule is that W. Johnson gets a new treatment in addition to his former treatment, possibly carrying its date. Therefore, the sequence of treatments is kept in a list for each patient, which may also be empty. Furthermore, two nodes are created that refer to the fact that the new treatment for the patient is based on a new diagnosis and needs a new prescription.

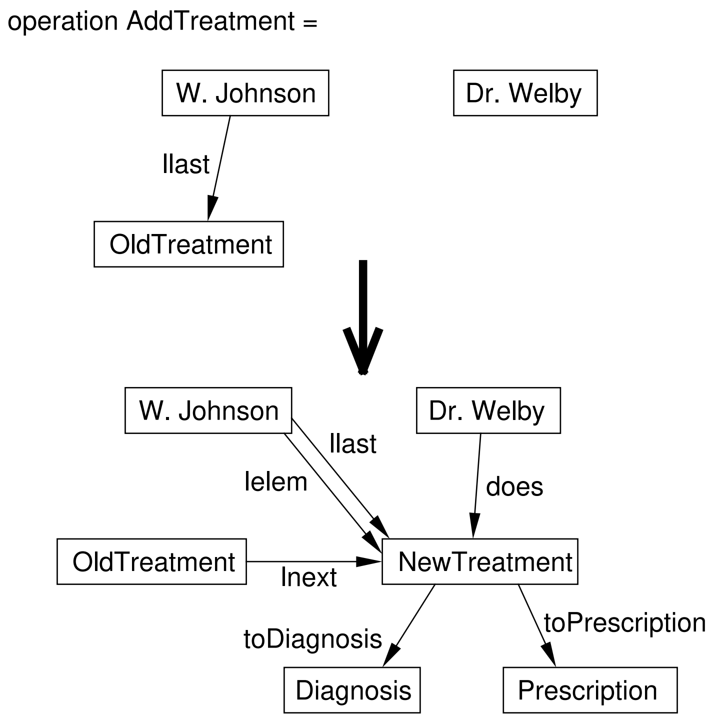


Figure 2: Informal outline for a typical operation in the diagnosis system

The system should support the doctor in finding a suitable series of examinations and tests based on the knowledge the system represents, in order to restrict the set of possible diseases of a patient.

We can also imagine that the user group of knowledge engineers works with the system. This group should be able to extend our knowledge base by further information, say symptoms and diseases as they are discovered, or to add additional tests for certain diseases as they are invented. This concludes the outline of the capabilities of our medical diagnosis system. We now turn to its formal description.

4 Developing a Graph Schema

We have just sketched some basic understanding of how and in what situations the medical diagnosis system could serve us. The next step is to analyze the given examples of our application and to put the results together in a graph schema.

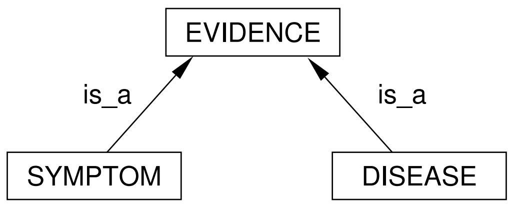
PROGRES is a strongly typed language, i. e. nodes, their attributes, and edges between them belong to exactly one type which determines their behavior.

The concept of typing nodes in PROGRES consists of two levels. Every node belongs to exactly one node type and each node type itself has a type, a so-called node class:

```
node type Doctor : DOCTOR end;
   (*  node class DOCTOR is the type of node type Doctor, which in turn is the type of
       nodes like Dr. Welby *)
```

This gives us the opportunity to treat types as first order values, i. e. they can be values of variables, parameters, and attributes and introduce subtype as well as parameterized polymorphism into the language. The characteristics of nodes, e. g. attributes and incoming or outgoing edge types, should be established at the highest possible level, i. e. in the node classes. On the other hand, instances are only allowed for nodes being of some type. The design of a (hierarchical) type schema is well-known in the areas of object modelling or database design.

Another important quality of the PROGRES graph schema for modelling applications is the possibility to build hierarchies with inheritance on node classes. Common properties of nodes belonging to different classes can be combined by creating a common superclass. As already mentioned before, diseases and symptoms can hardly be distinguished and thus we use the common superclass EVIDENCE for both.



The structure and the dependencies of a graph are given by its edges. The declaration of their types specifies to which classes the source and target node, resp., must belong to.

```
edge type anEdge : aSourceClass -> aTargetClass
```

The requirements for the classes of source and target nodes can be checked statically for every graph modifying operation. Considering a sequence of edge traversing operations, for every traversal step we can determine its static result class and check if this result class is a legal start class for the subsequent traversal. Thus, it can be assured that a traversal sequence at least potentially delivers a result.

Additional to the type of edges we are able to give information about the cardinality of source and target, e. g. the declaration

```
edge type does : DOCTOR [1:1] -> TREATMENT[0:n];
```

expresses the fact that a doctor may give zero or more treatments, but a treatment can be done only by exactly one doctor. Cardinality details can be omitted, the default is [0:n]. Cardinality information is useful for both static and dynamic consistency checking. Operations can be checked whether they use edges correctly, i. e. only as many edges may be incident to the same node as is allowed for this edge type. Of course, there is no way to check all cardinality conditions by analyzing the operations statically, because edges may be deleted or inserted or redirected explicitly, or edges may disappear with the deletion of an incident node. But for the sake of expressive power we gave up static type safeness at this point. Rather, cardinality may be used as an assertion by the specificator to the structure of his model. In case of its violation, an error is reported by the runtime system.

Apart from structuring the graph with nodes and edges, the node classes can be structured themselves. PROGRES offers attributes for nodes, these can be either of a built-in type (boolean, integer, or string) or imported. With the latter ones, types and resources that are implemented outside of PROGRES can be accessed. There are three different sorts of attributes: intrinsic, derived, and meta. Note that attributes are inherited from the father classes.

Meta attributes are read-only for operations, i. e appropriate values can be assigned in the declaration of the corresponding type once and forever. The price of a certain medicament may serve as an example for meta attributes.

Intrinsic attributes can be read and set during graph modifications. They serve as local variables of instances of a node. The name of a person or his date of birth are examples for this kind of attributes.

The third sort of attributes in PROGRES are derived attributes. Their evaluation may depend on other attributes of any nodes in the graph. Consider the simplified model of the income of a doctor:

```
node class DOCTOR
   derived
      Income : integer = sum( self.-does->.Costs );
end;
```

Assuming that every treatment has assigned costs (that may be fixed or derived themselves), the Income value of a doctor is computed by summing up the costs that all treatments cause to patients, for all treatments the doctor prescribes.

Derived attributes are well-known in the context of attributed grammars. Their evaluation follows a lazy strategy, i. e. read accesses of the attributes force all attributes they depend on to be evaluated. Once an attribute is evaluated, its value is stored. As long as the attribute is neither directly nor indirectly affected by modifications of the graph, the value remains valid. Changes of the underlying graph set the values of all potentially affected attributes to invalid and, therefore, force reevaluation for subsequent accesses. This offers an elegant way to derive values directly from the graph structure.[4]

Concluding this section we want to point out that we do not need to bother with syntactic details during the design of the graph schema, since PROGRES offers the facility of a syntax-directed editor and, moreover, the powerful type analysis of PROGRES notifies the specificator of any inconsistency of his produced graph schema. Furthermore, PROGRES supports graphical editing of the graph schema in an ER-like notation.

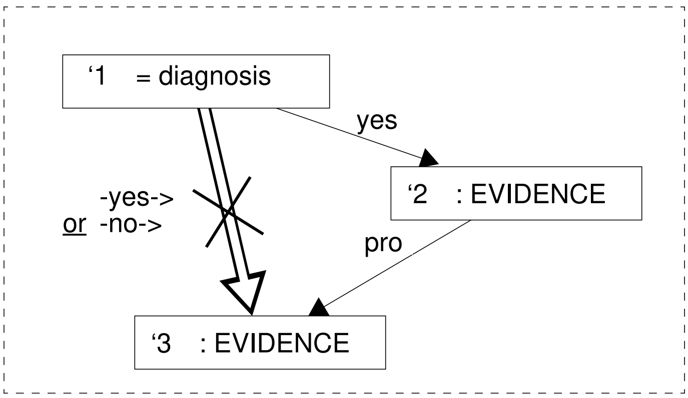
5 Specifying Graph Transformations

The last section dealt with the static portion of the medical diagnosis system. The next step is to model the operational behavior with graph rewriting rules and transactions offered by PROGRES. At this point we should better not care about efficiency aspects but rely on the declarative, rule based idea of visually modelling graph modifications with PROGRES.

Productions and tests are the basic operations of the PROGRES language for specifying graph manipulations and queries. Their graphical notation makes up the characteristics of the language and distinguishes it from most existing specification languages.

Tests offer the possibility to formulate queries in a visual manner, i. e. the underlying graph can be searched for a particular graph pattern. The test succeeds exactly when the pattern can be matched. Figure 3 shows a simple graphical test.

```
test GetProConjecture ( diagnosis : DIAGNOSIS ; out evidence : EVIDENCE )
   =
```




```
   return evidence := ‘3;
end;
```

Figure 3: Test GetProConjecture.

In this test we assume that we have already found a diagnosis ‘1, given as first parameter, and we confirmed an evidence ‘2. We are now looking for an evidence ‘3 that is required by

4. Unfortunately, the very elaborated attribute evaluation strategies known from compiler construction theory can not be used for general graphs since they rely on a dominating tree structure. For details on the incremental attribute evaluation strategy used in PROGRES cf. [KSW93].

the already found evidence for further restricting the possible diseases and which have not been proved or disproved, yet. If such a subgraph can be found in the underlying graph, then the test succeeds and returns the requested evidence node. You can see in this example that a (negated) path condition allows to define complex graph traversals. Path conditions can be build using sequences, alternatives, and iterations of traversal operations, and of (recursive) calls to other named path conditions.

This example also shows the use of parameters which is the same for transactions and productions explained later on. We strictly distinguish in and out parameters. in parameters may only be read whereas out parameters may only be written, consequently there are no transient parameters.

Generally, a graph pattern may consist of simple nodes (‘1, ‘2 and ‘3 in Figure 3), simple edges (yes and pro), and path conditions (-yes-> or -no->). A node in a graph pattern is labeled with an identifier and its type. Alternatively, a node may be passed as parameter, then the node is labeled with the formal parameter name. Edges are marked with their type, paths are given by their condition. Paths can also be declared separately, then they can be used just by their name. Negations of simple graph patterns can be expressed by crossing out the corresponding symbol.

There are some further possibilities to refine graph patterns which are outside the scope of this paper, such as restrictions for nodes, optional nodes, and node sets. The interested reader is referred to [Zü95]. In addition to structural patterns, node attributes in patterns can be restricted. For this reason, tests and productions have a condition part which allows stating boolean conditions for the attributes of matched nodes.

Speaking of pattern matching we should explain when we expect a pattern to match against a graph. PROGRES searches for subgraphs in the underlying host graph that have an one-to-one correspondence with the given pattern. It is also possible to match more than one node of the pattern with the same node in the graph, i. e. to perform a homomorphic map of the pattern onto the graph. This can be achieved by grouping the nodes together explicitly in a homomorphic clause.

Tests serve two different purposes, first as guard of a conditional clause that searches for a pattern in the graph and then branches on success or failure, second as a query that returns some nodes that match the pattern of the test for further use.

Finally, we turn our interest towards the graph rewriting rules offered by PROGRES. The so-called *productions* not only allow to perform queries but also to describe graph modifying operations on the graph. They consist of two parts. The left-hand side represents the pattern that is searched for and is very similar to tests. But furthermore, the right-hand side describes a replacement for the found pattern in the graph. Productions are well-known in the context of string and graph grammars.

In PROGRES, graph rewriting rules offer identical replacement for nodes, i. e. the properties of the identically replaced nodes are retained, especially their connection to the rest of the graph, called *embedding*. Identical replacement must be made explicitly by identifying two nodes, one on the left and one on the right side of the rule. Elements of the left side of the production that do not appear on its right side are deleted, whereas nodes and edges of the right side without a counterpart on the left are newly created. Embeddings for nodes can also be explicitly changed, values can be assigned to attributes, and values or nodes can be returned by the production.

Figure 4 shows our notion of how the system could propose to a doctor the suitable treatment for his patient. Suppose the doctor found an disease for which several treatments are known.[5] The doctor must now be sure that no other evidence is present for the patient which excludes any of these treatments. This is expressed by the crossed out node of the class EVIDENCE. Note that the doctor prescribes all feasible treatments to his patient.

We already mentioned that the intuitive graphical notation in PROGRES introduced above is one of its main characteristics. This is the reason that we do not explain its control structures (as e. g. conditions, iteration and sequencing) allowed in so-called *transactions* here which are rather similar to those in imperative programming languages.

The static type checker of the PROGRES environment assures the proper use of nodes, attributes, and edges with respect to their types in the specified operations. Relying on a statically correct specification we now want to verify whether the specification really "does" what we had in mind, i. e. if it describes the system that we intended to model.

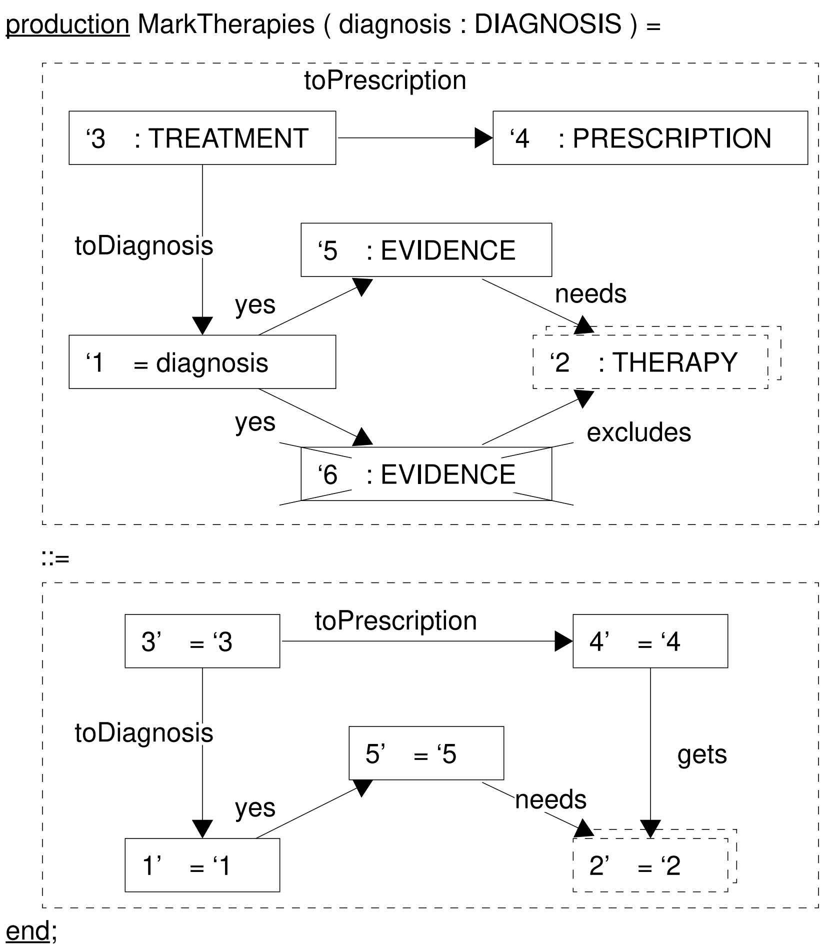


Figure 4: Production MarkTherapy

5. The dashed double box expresses a set of nodes.

6 Validating the Specification's Correctness

Assume that we have specified all the static and the dynamic aspects of our medical diagnosis system. The PROGRES interpreter offers us the opportunity to execute the specification and watch the effects of the graph rewriting operations. An independently running graph browser may connect to the underlying host graph and show its state change during a graph operation. Thereby, the user can easily control if a graph rewriting rule matches the expected pattern and if it does the proper replacement. The interpreter has debugger abilities. If the specificator gets aware of unexpected behavior he can proceed by stepping through productions and check actual matches of its patterns. He may even undo arbitrarily long sequences of graph replacements step by step and replay them as needed. The reader is referred to [Zü92, Zü95] for details concerning the implementation of the interpreter. Figure 5 shows a snapshot of a sample session with the PROGRES environment and the interpreter running. The currently used graph browser is called *EDGE*, a development from the University of Karlsruhe (cf. [Ne91]).

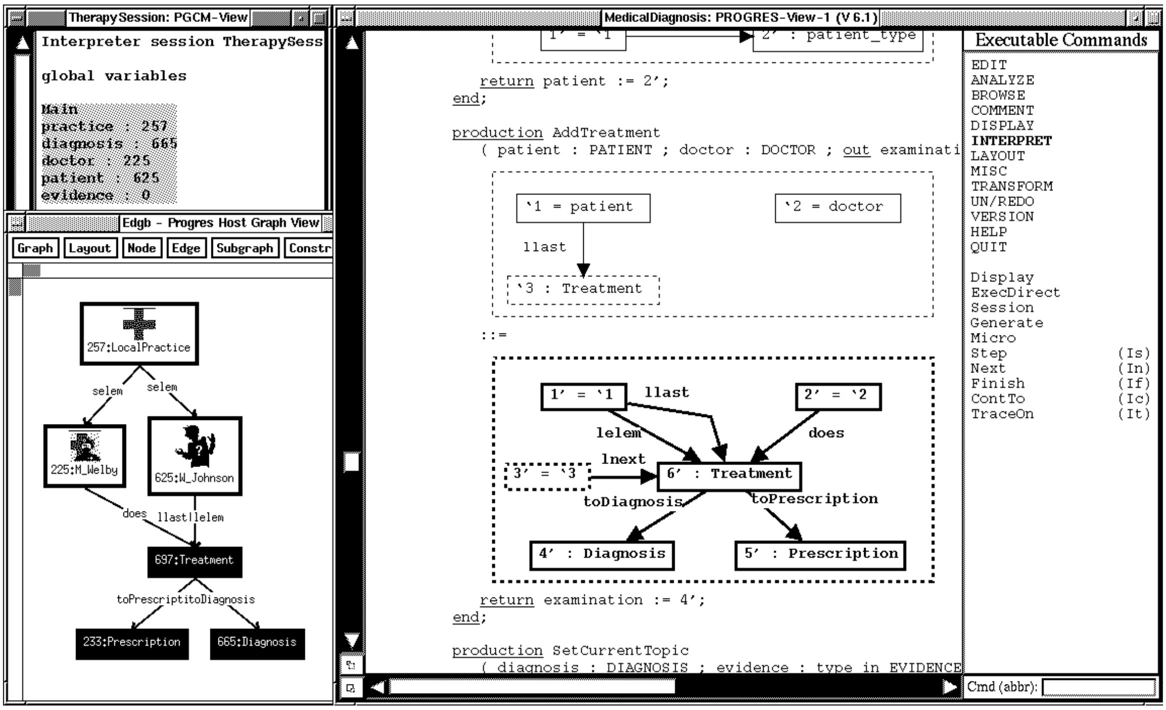


Figure 5: Sample session with the PROGRES interpreter

We are now at a point where we can consider both the static and the dynamic aspects of our specification correct with respect to our intended model. But there is still room for improvement concerning the response time of the system.

7 Improving Efficiency

Based on the validated description of the modelled system we are interested in its improvement. We have derived profit from the declarative specification but eventually we are aiming for an efficient application.

The most time-consuming portion in the specification is the graph pattern matching process. It can be easily imagined that a search through the whole host graph is not the most efficient way of accessing some node or matching a subgraph. The search for a certain node can be directed by restricting the graph search to some area within the graph. There are two ways to do this. The first is to give nodes as parameters. Alternatively, the specificator may extend his specification with additional structures, e. g. nodes of a special and unique marker type. Both possibilities provide the graph search engine with anchor points for the match. This works only in the case that nodes or structures to search for are known in advance. If we are searching for nodes with certain attribute conditions, we take advantage of the underlying graph storage system which offers methods for indexing attributes of nodes. This remarkably accelerates the access to the matching nodes.

Concluding the possibilities of efficiency improvement we want to mention the materialization of paths, i. e. paths may be declared static (well-known efficiency improving database technique [KeMo92]). The effect is that paths matching the declaration will be represented by internal edges which makes their traversal more efficient. The management of static paths itself can be time-consuming when the graph is frequently changed. This means the designer of the specification has to keep in mind the trade-off between maintaining and traversing (static) paths.

8 Generating a Prototype

Based on the specification which is hopefully correct and more or less efficient, a prototype can be built. It is based on the source code generated on demand in the PROGRES environment and on a standard frame architecture for interactive systems. The prototype offers a graphical user interface for interactively executing productions and transactions. An entry in the production menu is created for selected productions in the specification. If the chosen production needs input parameters, a dialog window appears for entering them either in the form of simple values or as references to global variables. Their values are permanently shown in a variable view.

Future plans are to divide the set of productions according to different user groups. We have the notion of people on different jobs who may have quite different demands on the system. In our medical diagnosis system we can imagine the nurse creating files for new patients, the doctor prescribing medication depending on the history of the patient, and the knowledge engineer who may introduce new treatments for known diseases or even new diseases. They all have special views on the situation which should be met by interfaces (views), created especially for their needs and restricted according to their permissions.

Furthermore, the user is capable of observing the host graph by means of a browsing tool. One can easily adopt the standard browser to his special application using the capabilities of the underlying toolkit *Tcl/Tk* (cf. [Ous94]). The different tools of the prototype are integrated by using common state information, shared access to the host graph, and notification of graph modifications based on events. For example, the execution of a production having result parameters has several effects: First, it updates the variable view with the new values, then the graph browser recognizes the result nodes and marks them, and finally the actual parameters of the production are stored for repeated execution. The functionality of the browsing tool is extended by an input mechanism that allows to select nodes that are passed directly to the parameter request dialog of the next chosen production.

The prototype has full undo/redo ability, i. e. all the performed choices and executions are recorded and the user may undo them, even the graph modifying steps. Since the created prototype is based on the functionality of the underlying graph storage system GRAS [KSW93] (just like PROGRES itself) we have persistence of data structures, recovery and multi-user capabilities.

A snapshot of a possible scenario from the doctor's point of view is presented in Figure 6.

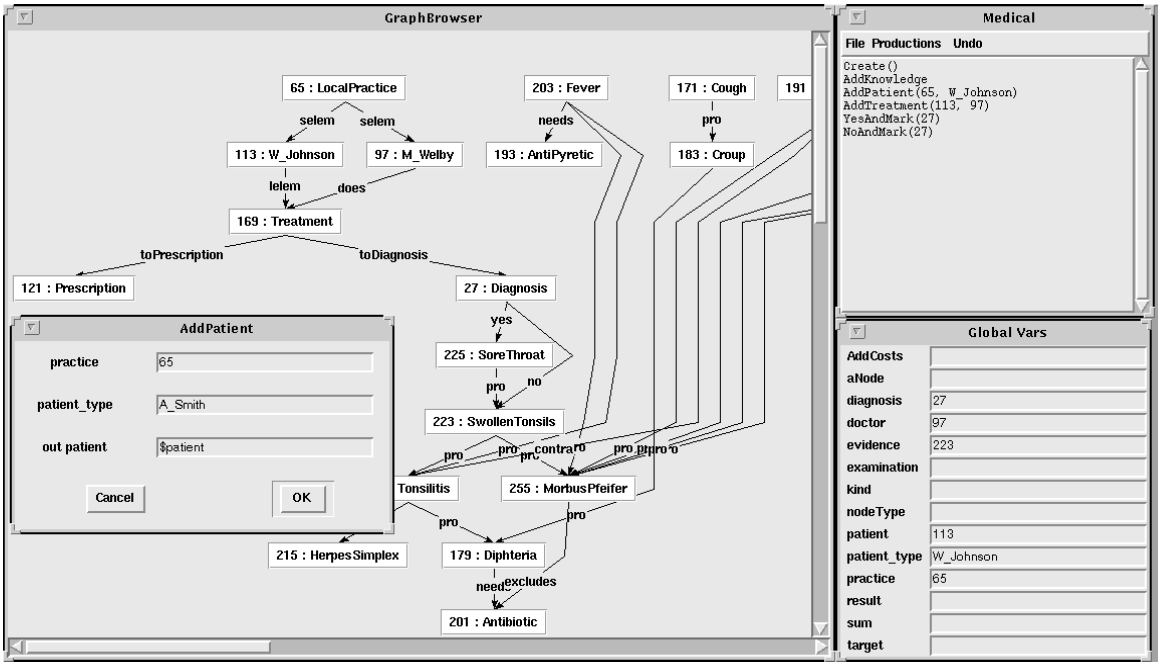


Figure 6: Prototype for the Medical Diagnosis System.

We now have reached a point where a stand-alone, automatically generated prototype can be presented for a given specification based on a standard system architecture.

9 Deriving an Implementation

The prototype is a good basis for further considerations concerning the final application. Appearance and behavior of the prototype can be studied and compared to the needs of the different user groups [Hal90]. It is not within the scope of this paper to discuss the stages of creating an application, but we want to mention that the PROGRES compiler produces easy-to-read C or Modula-2 code from which a final implementation can be derived by adding new features like operating system calls or user interface functionality or by performing optimization steps. Finally, the code can be integrated as an encapsulated abstract data type implementation into the architecture of the overall system.

Until now, the final implementation is still built on top of the database system GRAS and has therefore the same characteristics as the previously described prototype. We are currently planning to add a new compiler back-end to the PROGRES environment which generates stand-alone C++ code based on a main memory graph implementation. Thereby, we hope for a speed up of 1000 times payed for by dropping persistence, multi user access, backtracking and undo/redo abilities.

10 Conclusion

Graph grammar engineering is a new methodology for the development of information system components. It starts with an informal requirements definition and produces an executable specification with precisely defined semantics. A whole suite of tools performs about 300 different consistency checks in the specification, supports intertwined editing and debugging, and generates rapid prototypes of specified systems. These prototypes are built on top of the nonstandard database system GRAS and have Tcl/Tk-based user interfaces.

Compared to [ELNSS92], the graph grammar engineering approach presented in this paper is no longer restricted to abstract syntax graphs used in CASE environments but offers a methodology for developing very general complex data structures. Many features concerning the expressiveness and efficiency of PROGRES has been added to the language. The first time automatic code generation and the generation of stand-alone prototypes is available. These new abilities of the language and its environment now provide the necessary support for our more general graph grammar engineering methodology.

To summarize, the language PROGRES, its tools, and the presented graph grammar engineering methodology are a first step into the direction of establishing graph rewriting as new specification and programming paradigm. Nevertheless, we have to admit that currently available tools and techniques are far from being as mature as (for instance) logic-oriented or functional programming centered tools and techniques. A lot of research is still necessary to improve efficiency of graph rewriting, to replace the data model of flat graphs by a new data model of hierarchical graphs (cf. [CCM92]), and to introduce an urgently needed module concept. Hierarchical graphs as well as a module concept are strictly necessary for the development of large systems instead of relatively small and uninteresting information system components.

The current prototype of the PROGRES environment is available as free software and comprises about 400 000 lines of Modula-2 and C code. The release of PROGRES is available via anonymous ftp from the following site:

ftp-i3.informatik.rwth-aachen.de in directory /pub/PROGRES

References

[BFG95] D. Blostein, H. Fahmy, A. Grbavec: *Practical Use of GraphRewriting*; in [GG95].

[Boo94] G. Booch: *Object-Oriented Analysis and Design*; Benjamin/Cummings, Redwood Cit, CA (1994).

[CCM92] M. Consens, I. Cruz, A. Mendelzon: *Visualizing Queries and Querying Visualizations*; ACM SIGMOD Record 21, no. 1 (1992), pp. 39-46.

[ELNSS92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, A. Schürr: *Building Integrated Software Development Environments Part I: Tool Specification*; ACM Trans. on Software Engineering and Methodology 1, no. 2 (1992), pp. 135-167.

[ELS86] G. Engels, C. Lewerentz, W. Schäfer: *Graph Grammar Engineering: A Software Specification Method*; in [GG86], pp. 186-201.

[GG78] V. Claus, H. Ehrig, G. Rozenberg (eds.): *Proc. Int. Workshop on Graph Grammars and Their Application to Computer Science and Biology*; LNCS 73, Springer-Verlag, Berlin (1978).

[GG82] H. Ehrig, M. Nagl, G. Rozenberg (eds.): *Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science*; LNCS 153, Springer-Verlag, Berlin (1982).

[GG86] H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (eds.): *Proc. 3rd Int. Workshop on Graph Grammars and Their Application to Computer Science*; LNCS 291, Springer-Verlag, Berlin (1986).

[GG90] H. Ehrig, H.-J. Kreowski (eds.): *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*; LNCS 532, Springer-Verlag, Berlin (1990).

[GG95] H. Ehrig, G. Engels, G. Rozenberg: *Proc. 5th Int. Workshop on Graph Grammars and Their Application to Computer Science*; Springer-Verlag, Berlin (to appear).

[GGN90] H. Göttler, J. Gönther, G. Nieskens: *Use Graph Grammars to Design CAD-Systems*; [GG90], pp. 396-410.

[Hal90] M. Hallmann: *Prototyping of complex Software Systems*; (in German) Teubner-Verlag, Stuttgart (1990).

[Him90] M. Himsolt: *GraphEd: An Interactive Graph Editor*; STACS, LNCS 349, Springer-Verlag, Berlin (1990).

[KeMo92] A. Kemper, G. Moerkotte: *Access Support Relations: An Indexing Method for Object Bases*; Information Systems 17-2, Pergamon Press (1992), pp. 117-145.

[KSW93] N. Kiesel, A. Schürr, B. Westfechtel: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*; Proc. 6th Int. Workshop on Computer-Aided Software Engineering, IEEE Computer Science Press (1993), pp. 272-286.

[LöBe93] M. Löwe, M. Beyer: *An Implementation of Algebraic Graph Rewriting*; Proc. 5th Int. Conf. on Rewriting Techniques and Applications, LNCS 690, Springer-Verlag, Berlin (1993), pp. 451-456.

[NaSc91] M. Nagl, A. Schürr: *A Specification Environment for Graph Grammars*; [GG90], pp. 599-609.

[Ne91] F. Newbery: *The Design of an Extendible Graph Editor*; Dissertation, University of Karlsruhe, Germany, LNCS 704, Springer-Verlag, Berlin (1991).

[Ous94] J. Ousterhout: *Tcl and the Tk Toolkit*; Addison-Wesley, Reading, MA (1994).

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*; Prentice-Hall, Englewood Cliffs, NJ (1991).

[Sch91a] A. Schürr: *Operational Specification with Programmed Graph Rewriting Systems*} (in German), Deutscher Universitäts-Verlag, Wiesbaden (1991).

[Sch91b] A. Schürr: *PROGRESS: A VHL-Language Based on Graph Grammars*; [GG90], pp. 641-659.

[Zü92] A. Zündorf: *Implementation of the Imperative/Rule Based Language PROGRES*; Technical Report AIB 92-38, RWTH Aachen, Germany (1992).

[Zü94] A. Zündorf: Graph Pattern Matching in PROGRES; [GG95].

[Zü95] A. Zündorf: *A Development Environment for PROgrammed Graph REwriting Systems*; (in German), Dissertation, RWTH Aachen, Germany (to appear 1995).