Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

Andreas J. Winter and Andy Schürr

Lehrstuhl für Informatik III
RWTH Aachen
52056 Aachen, Germany

Specification and rapid prototyping of graph manipulating software by means of PROgrammed Graph REwriting Systems (PROGRES) is a paradigm, which attracts more and more interests in various fields of computer science. Nowadays produced specifications for process modelling tools, database query languages, or Conceptual Graph manipulations have a typical size of about 100 printed pages. They suffer severely from the lack of any graph rewriting system module concept. As a first remedy, we have just developed a module concept in the flavor of Modula-2 for our graph rewriting language and system PROGRES. It supports a variety of programming-in-the-large styles, including the construction of Abstract Graph Types. Having made the experience that "full data abstraction" is in conflict with the visual programming paradigm of PROGRES, we are now starting to add active database system concepts for the construction of updatable abstract graph views.

Keywords: programmed graph rewriting, visual language, module concept, views

1. Introduction

Graphs play an important role within many areas of applied computer science, and there exists an abundance of visual languages which have graphs as their underlying data model. Prominent examples drawn from the areas of Software and Knowledge Engineering are Entity Relationship Diagrams, Statecharts, Petri Nets, and Conceptual Graphs. 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. First examples of (production) rule-based expert systems, like DENDRAL [14] or GPS [28], are more than 30 years old.

Why not combining two successful principles in one formalism? This is the common idea of the graph grammar research community. At the beginning research was mainly focused on producing theoretical results due to the limitations of available computer hardware. This situation has changed gradually with the appearance of graph grammar based tools like

- GraphEd [19], which supports mainly graph layout and editing,

- PAGG [15], which is a no longer available graph grammar editor and interpreter,
- DiaGen [23], which is mainly used for generating visual language editors and parsers,
- AGG [22], which serves as a learner's environment for category theory based graph rewrite rules, or
- PROGRES, which is a graph grammar programming language and environment for all above mentioned activities (except the construction of pushouts).

The language PROGRES comprises rule-based, object-oriented, and imperative programming constructs. It is used at various sites for specification and rapid prototyping activities (for configuration management, process modeling, visual database query languages, database reverse engineering, and analysing distributed system traces). Most of these graph grammar specifications have a size of about 100 pages. Keeping these specifications in a consistent state and reusing generic parts of one specification within another specification is a nightmare without the existence of any module concept.

These problems should be familiar for software developers of the late 60ies or expert system developers of the late 70ies. Well-known software engineering concepts like "abstract data types" [29] and "programming–in–the–large" [9] have been invented to overcome these problems. Later on, these concepts have lead to the development of modular programming languages like Modula-2 [40] and Ada [37], the development of software design languages like HOOD [30] or EMIL [4, 24], and finally to the development of module concepts for knowledge base representation languages like EPSILON [7] and PROTOS-L [2].

Rather recently, a number of graph grammar researchers joint their efforts in the so-called GRACE initiative [1]. Their common goal is the development of a GRAph Centered Environment, which supports modular programming with various forms of graph data models, rewrite rules, and rule regulation mechanisms. Related publications are first attempts to introduce import/export as well as inheritance/refinement relationships into the world of graph grammars [10, 17], to encapsulate graph transformation algorithms by means of so-called transformation units [20, 33], to adapt the concept of database views to graph data types [26], and to develop new concepts for the design of hierarchical graph structures [12] or distributed systems of graph objects [36].

Taking all these sources of inspiration and the state-of-the-art into account, we are currently developing a *graph grammar module concept* for the language PROGRES, which is intentionally kept as simple as possible. Its first version, presented here, is very similar to the module concepts of Modula-2 and Ada. Modules (packages) of these languages are just encapsulated groups of type and procedure declarations. They have no runtime semantics at all, but restrict the visibility of declarations at program development and compile time (cf. [39]).

The main advantages of such a module concept are as follows:

- It is easy to use and solves nevertheless the most urgent problems of the "PROGRES language user community".
- Its modules may be used for quite different purposes, varying from the realization of encapsulated abstract data types to the definition of open graph database (sub-)schemata.
- Its implementation can be accomplished within a short period of time.

Long lasting efforts are still necessary to develop all the details of more ambitious graph grammar module concepts and to combine them within a single coherent graph grammar language.

The paper gives first a rather condensed introduction to the language PROGRES by introducing a specification for a special form of balanced binary trees, so-called red/black trees. Then we will present the selected module concept by identifying an interface for the red/black tree specification in the sense of an abstract graph type. Based on this suggestion we will show how modules may be used (imported) in other modules to construct new abstract graph types on top of already existing ones. Finally, we will describe how we realize updatable graph views by means of active constraints.

2. Specification–in–the–Small with Graph Rewrite Rules

The purpose of this paper is neither to introduce the language PROGRES in full detail nor its integrated development environment. For these issues the reader is referred to [31, 41, 21]. In this section we try to give an impression of the expressive power of the language PROGRES by introducing a suitable example application, the specification of red/black trees [5].

The language PROGRES is based on the data model of directed attributed graphs and allows to describe transformations on graphs by rules declaratively. It was originally designed for describing internal data structures, operations and relations of tightly integrated Software Engineering tools [25]. Its semantics is formally defined, and it is supported by an integrated set of tools which allow to edit graphically or textually, to analyze, and even to execute specifications. Therefore, PROGRES constitutes an executable visual language for specifying systems based on internal graph structures [35]. Together with the ability to generate stand-alone prototypes from specifications we have already demonstrated the suitability of our *Graph Grammar Engineering* approach with PROGRES for some application areas (cf. [11, 34]).

The example in this paper stresses the visual character of graph rewrite rules rather than the usefulness of graphs as an underlying data model. Red/black trees have been selected in order to demonstrate the advantages of PROGRES in contrast to conventional programming languages. The data model of red/black trees is a balanced binary search tree with colored nodes. Rather complex restructuring operations are invoked from time to time to rebalance a red/black tree after some insertion or deletion.

PROGRES specifications consist of two parts: the static *graph schema* and the *schema consistent operations*. The schema allows to express static properties of the regarded class of graph. Its visual notation — not shown here — has the form of EER diagrams with entities called node types and binary relationships called edge types. Properties of node types, which are not relationships between nodes, are modeled as attributes. A type hierarchy, similarly to object-oriented approaches, can be established on these node types. Defining edge types in the schema allows to check statically whether edges are used in a correct context (source and target nodes) in graph transformations. Furthermore cardinalities in the schema and constraints reflect the invariants of the modeled data structure e. g. by restricting the number of source or target nodes that can be reached by traversing edges of certain types.

```
node type RBTREE end;

edge type Root : RBTREE [0:1] -> RBNODE [0:1];

node type RBNODE
  constraint
    CorrectColored
      = self is not with (( <-RedLeft- or <-RedRight- )
                         and ( -RedLeft-> or -RedRight-> ));
end;

edge type BlackLeft : RBNODE [0:1] -> RBNODE [0:1];

edge type BlackRight : RBNODE [0:1] -> RBNODE [0:1];

edge type RedLeft : RBNODE [0:1] -> RBNODE [0:1];

edge type RedRight : RBNODE [0:1] -> RBNODE [0:1];

node type ELEM
  intrinsic
    key Value : integer;
end;

edge type Contains : RBNODE [0:1] -> ELEM [1:1];
```

Figure 1: Cutout of the red/black tree graph schema

The textual schema in figure 1 defines the properties of the red/black tree data structure as follows:

1. A red/black tree is represented by a placeholder node of type `RBTREE` and contains only nodes of one type `RBNODE`. The root of the tree is identified by an edge of type `Root` starting at the placeholder node of the tree to its actual root node.

2. A red/black tree contains different types of edges[a]. Any parent/child edge is either red or black and leads either to a left or a right child (cf. corresponding declaration of edge types).

[a] Note that we do not use colored nodes as in [5] but colored edges, instead.

3. Any node has at most one left and right child and at most one parent node (cf. cardinalities in edge type declarations).

4. The target of a red colored edge is never the source of another red colored edge, thereby ensuring balanced trees (cf. constraint `CorrectColored` for `RBNODE`).

5. Instead of identifying different nodes in a red/black tree by integer key values themselves, we establish a `Contains` relation from each internal tree node `RBNODE` to the concrete element node of type `ELEM`, which has a unique integer as key attribute `Value`.

Having defined the static graph schema we are now able to develop the transformation rules on red/black trees. To guarantee that the tree remains balanced, reorganization of a given red/black tree may become necessary after insertion or deletion of nodes. The "backbone" of the reorganization process are four rotation operations. We neither explain the operations in detail nor do we prove their correctness with respect to the outlined invariants. Instead, we want to present an example transformation rule which accesses graph elements directly and therefore describes the rotation operation in a visual and intuitive way.

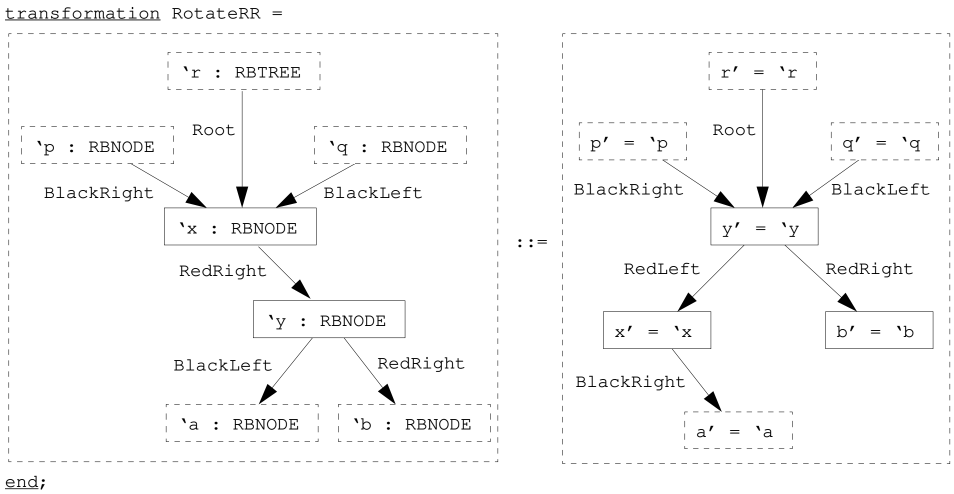
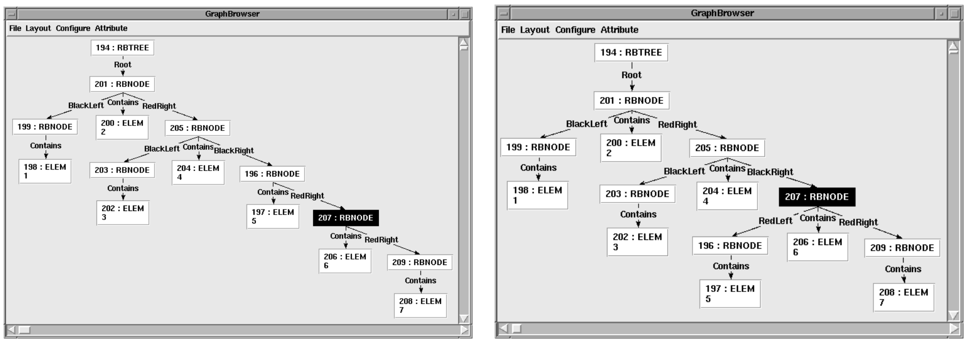


Figure 2: Rotation operation on red/black trees

Generally, the semantics of a transformation rule is to find a matching subgraph according to the pattern in the rule's left side and to replace it with its right side The given rule in figure 2 is adopted from the description in [5] and applies to an underlying working graph containing a subgraph "according" (i. e. isomorphic) to its left side. In this example the pattern in the left side will match, if a node can be found with one incoming and — at the same time — one outgoing edge of type `RedRight`. Since a graph pattern like that would violate the constraint for nodes of type `RBNODE` in figure 1 (see item 4 in the property list above), we have to perform a reorganization operation on the tree and redirect some of the tree edges yielding a proper red/black tree with respect to our invariants. For performing the necessary edge redirections we do not only need to search the graph for a "bad" node in the above described sense, we also need its context. This allows to specify the necessary redirection of the affected edges explicitly in the rule rather than programming the redirection of pointers in a usual programming language. Note that in this rule nodes are neither inserted nor deleted during its application, they are all preserved. This is denoted by the node labels of the form `n'='n` in the rule's right side. The meaning of the dashed boxes in the rule is that they may or may not have a counterpart in the underlying working graph. This allows to handle several special cases in one rule. For example node `x` is either the root of the tree or it is the (left or right) son of one of the nodes `p` and `q` (of course not of both, though). In all cases the incoming edges have to be redirected to the node `y` which becomes root of the former `x` subtree. Figure 3 shows a red/black tree just before and after the transformation with the given rule. The highlighted node shows which node is matched by the node `y` in the rule.

This example was chosen to show that graph transformation rules are an adequate means to describe the restructuring operations on tree- and graph-like data structures. We do not want to discuss the specification of red/black trees with PROGRES in further detail. The complete specification can be found in [32].



a) Before rule application b) After rule application

Figure 3: Application of `RotateRR` (PROGRES system screen dump)

To summarize, PROGRES and its integrated environment are a first attempt to define and implement a graph-transformation-based, strongly typed programming language with well-defined syntax, static and dynamic semantics. Being a mixed textual and diagrammatic language, it permits quite different styles of programming, and supports

- graphical as well as textual definition of graph schemata (cf. figure 1) with declaration of derived graph properties,
- rule-oriented and diagrammatic specification of atomic graph transformation

steps by means of parametrized graph transformation rules (cf. figure 2) with complex preconditions, and

- imperative programming of composite graph transformation processes by the means of deterministic and nondeterministic control structures.

For a detailed discussion of the language's features the reader is referred to [31, 21].

3. Specification–in–the–Large with Abstract Graph Types

In the last section we have demonstrated the usefulness of graph transformation systems for describing operations on complex graph-like data structures. In fact, many PROGRES specifications have been written in the past describing complex graph transformations for given application areas. But, until now there was neither the possibility to abstract from technical implementation details nor to reuse specifications as service providers like in other programming languages. In order to close this gap we will now focus on the question how to handle different specifications from an architecural point of view. Therefore, we adapt the concept of data abstraction, which is well-known and widely studied in the software engineering community, to specifications based on graphs by introducing *abstract graph types* (AGTs) and their access operations. Based on the practical experiences with classical programming languages we have developed a module concept for PROGRES specifications, distinguishing between their interface and their body, and established a concept for *specification–in–the–large* with abstract graph types.

In order to avoid misunderstandings about the term *module*, we will mention some characteristics which are important for our module concept:

- A module constitutes a logical unit of data together with corresponding operations of a certain complexity (abstract data type).
- A module provides other modules with resources at its interface hiding the implementation, which may itself use other interfaces in its body.
- A module can be developed, compiled, and tested independently from other modules and is replacable by a module with the same export interface.

Applying these ideas to our red/black tree specification, we want to use the specification `RBTREES` as an efficient implementation for operations on sets. The module `SETS` itself is supposed to provide applications with suitable set access operations in its interface not revealing the internal realization of sets by trees.

The first step in this direction is to define an abstract graph type `RBTREES` by identifying the parts of the red/black tree schema and the operations which should contribute to its interface. The schema presented in figure 1 was written having this intention mind. It can be easily divided into parts for internal realization of red/black trees and external access. Note that it is the purpose of the so far unnecessary placeholder node of type `RBTREE` to represent the tree itself. This makes it possible to encapsulate the internal tree nodes and edges in the body. Considering operations of red/black trees we have to extract adequate access headers for the interface. The example transformation introduced in figure 2 as a purely internal restructuring operation is certainly not to be called from outside the red/black tree module and should not be added to its interface. Since trees are a well-known data structure, we present the "natural" interface of the abstract graph type RBTREES without further discussion in Figure 4.

```
interface RBTREES
  from ELEMENTS import ELEM;
  node type RBTREE end;
  transformation RBTreeCreate( out T : RBTREE);
  transformation RBTreeRemove( T : RBTREE);
  query RBTreeIsEmpty( T : RBTREE);
  transformation RBTReeInsertValue( T : RBTREE ; Elem : ELEM);
  transformation RBTreeDeleteValue( T : RBTREE ; Elem : ELEM);
  query RBTreeValueExists( T : RBTREE ; Elem : ELEM);
end;
```

Figure 4: Interface of the adt RBTREES

The exported resources are rather evident. First we have the node type RBTREE representing the current tree. This type remains totally opaque in the interface. Furthermore we provide operations for creation and deletion of trees, for inserting and removing values, and, finally, query operations for testing emptiness and the existence of certain elements in the tree. The graph transformation and query operations are exported with their signatures, i. e. their names together with their lists of formal parameters. Note that we benefit from abstract graph types by importing external resources at the same time we are introducing them. We have decided that the abstract graph type ELEMENTS should not be internally hidden in red/black trees, but also offers its services to other clients. It exports the node type ELEM, which encapsulates the internal structure of its contents. Any element node has a key attribute Value which is used as the red/black tree index. As a consequence, it is forbidden that two ELEM nodes with equal Value attribute are part of the same red/black tree or the same set. ELEMENTS exports a function GetElemValue to read this attribute. We omit the further details of the interface and the body of ELEMENTS here.

The query operations RBTreeIsEmpty and RBTreeValueExists deserve an extra explanation. They do not have an explicit boolean result parameter telling whether the tree is empty or the respective element was found. Instead, the operations have a boolean character themselves: they succeed if the tree is actually empty or the element has been found in the tree and fail if the tree is not empty or the element is not present.

Note that the module concept presented so far is influenced by Modula-2 [40].

Consequently we have no means to ensure that a module's interface actually constitutes an abstract data or graph type. A module (interface) itself is no more than a collection of declared resources. It is the responsibility of its creator to provide suitable types and access operations with respect to the intended semantics of the module.

Having developed an abstract graph type for red/black trees, we are now ready to implement clients which can abstract from the internal realization and use the exported resources on a higher level. Of course, this client module itself should have again the character of an abstract graph type. Our example will be an AGT `SETS`, which hides its realization with trees in its body and provides the usual set access operations for further use.

Before presenting the `SETS` interface we have to discuss its internal realization. That means having the prerequisites for using other modules on a technical level, we must explain *how* we think that the usability relation is applied correctly for this concrete example on a conceptual level.

The first problem that has to be solved is the appearance of an appropriate node type representing the set. Although not visible for the user later on, we want to put this type into relation with the available tree type at the interface of `RBTREES` and to invoke the corresponding tree operations on it. Taking PROGRES specifications of the past, we see that the mechanism of *inheritance* on node types is widely used for this purpose and could be very naturally applied for our concrete example. That means in the body of the set graph type, we could realize the type `SET` as a subtype of `RBTREE`. In that case a node of type `SET` — being a special kind of `RBTREE` — would have available all the operations from `RBTREES` directly.

Although this approach is feasible from a technical point of view, we have chosen not to follow this approach for two reasons:

- In this solution inheritance is not used for subtyping with respect to the properties and behavior of the involved types, but merely for reusing the code which is related to the inherited type. A long discussion in the area of object-oriented modeling and implementation has taken place, whether this idea abuses the concept of inheritance. As a consequence, we refrain from using inheritance for this purpose.

- From the user's point of view we see another problem independent from any conceptual doubts. Consider the case that `SETS` is used in some other module and the type `SET` is specialized as `USER_SET` for further user specific implementation purposes. Note that the user does neither know nor care about the internal realization of `SET` being a subtype of `RBTREE`. In this case, he might extend "his" type with further attributes which may coincidently be present as attributes of `RBTREE`. In that way an inheritance conflict occurs for which the user cannot identify the cause without looking into the realizations of all involved types in the inheritance hierarchy. This contradicts the principle of data abstraction.

Having argued against the approach based on inheritance, we will now discuss the solution that we have preferred to use.

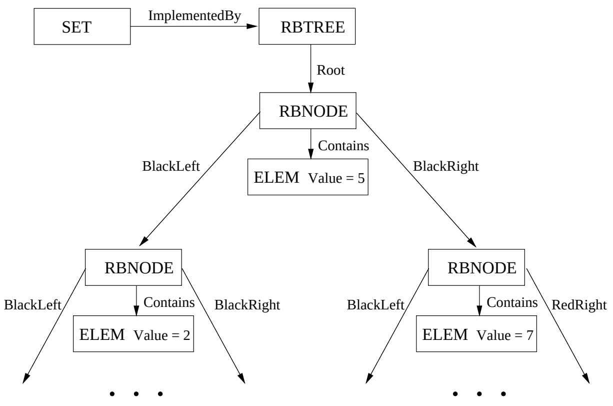


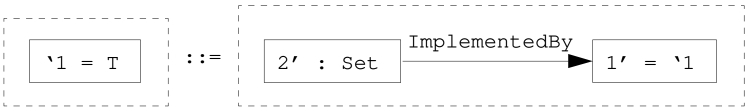
Figure 5: Realization of sets with red/black trees

Figure 5 show a small excerpt from an instance set graph. Instead of using inheritance to establish a relation between sets and their implementation trees, we model this relation with an explicit edge `ImplementedBy` from the anchor node representing the set to the anchor node representing the tree. With this realization operations on the set node can invoke the suitable tree operations on the tree node reached by traversing the edge. The advantage of this solution is that in the same way as the tree node of type `RBTREE` is an entry point for accessing the tree, the node of type `SET` constitutes an entry point for the set. Neither of them has to reveal anything about the realization of its internal strucure. For further imports from other modules only the type of the respective anchor node has to be exported at the interface.

In figure 6 we have sketched the body of the abstract graph type `SETS`. The `FROM ...IMPORT` clause introduces imported identifiers which can be used in the following as if they were locally declared. We have already seen an example of this kind, the import of the `ELEM` type. The set implementation needs to import (all) the available resources from the interface of `RBTREES`. This includes both, the type declaration (`RBTREE`) for the anchor of the tree and the operations.

The transformation rule `SetCreateImpl` demonstrates how the relation of the set and tree anchor nodes in the graph structure in figure 5 is constructed. It takes an existing tree node as parameter `T` and creates both the corresponding set node `2'` and the `ImplementedBy` edge to express that the implementation of this new set is based on the given tree. Furthermore, the transformation `SetInsertValue` shows how the corresponding tree operation `RBTreeInsertValue` is invoked after traversing to the tree anchor node for a given set anchor node. We use again the AGT `ELEMENTS` encapsulating the realization of the nodes to be inserted into the set. But note that we do not allow duplicate entries in the set (and tree) with respect to the element's key value.

```
body SETS

  from ELEMENTS import ELEM;

  from RBTREES import RBTREE, RBTreeCreate,
                     RBTreeInsertValue, RBTreeDeleteValue,
                     RBTreeValueExists;

  edge type ImplementedBy : SET [0:1] -> RBTREE [1:1];

  transformation SetCreate( out S : SET) =
    use T : RBTREE
    do
        RBTreeCreate ( out T )
      & SetCreateImpl ( out S, T )
    end
  end;

  transformation SetCreateImpl( out S : SET ;
                                    T : RBTREE) =
```




```
    return S := 2';
  end;

  transformation SetInsertValue( S    : SET ;
                                 Elem : ELEM) =
    (* Operation has to check whether value already
       exists to avoid multiple entries in set. *)
    choose
      SetValueExists ( S, Elem )
    else
      RBTreeInsertValue ( S.-ImplementedBy->, Elem )
    end
  end;

  . . .

end body;
```

Figure 6: Parts of the implementation of `SETS`

Based on the given implementation for sets in figure 5 and the concrete realization in the body in figure 6, the AGT `SETS` needs to provide an interface itself for further clients. Sets, like trees, are a well-studied data structure. Hence the determination of suitable interface operations is straight-forward. The export of the type `SET` representing the set is similar to the export of the type `RBTREE` in the interface of `RBTREES`. In general the set interface is similar to the tree interface as shown in 7.

The correspondence of the interfaces presented so far is reflected in the implementation of the `SETS` body, where almost no new internal operations have to be added. Essentially the operations in the body take their input parameters, which are sets, and pass them to the corresponding imported tree operations. On one hand, that allows to benefit from the efficient underlying tree implementation for the set example. But, on the other hand, there is no possibility to access internal

```
interface SETS
  from ELEMENTS import ELEM;
  node type SET end;
  transformation SetCreate( out S : SET);
  transformation SetRemove( S : SET);
  query SetIsEmpty( S : SET);
  transformation SetInsertValue( S : SET ; Elem : ELEM);
  transformation SetDeleteValue( S : SET ; Elem : ELEM);
  query SetValueExists( S : SET ; Elem : ELEM);
end interface;
```

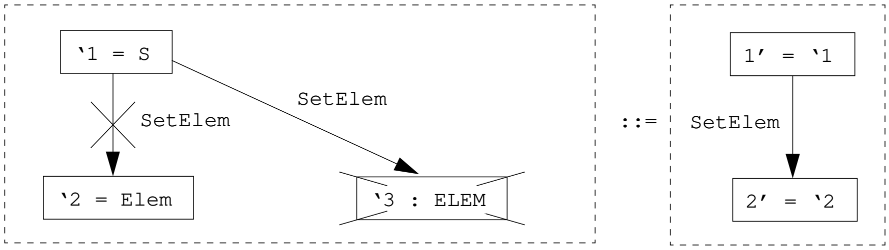
Figure 7: Straightforward SETS interface

set elements without using the interface of SETS. This *contradicts* the idea of *visual programming* with rules, which was our starting point. Although this solution might be satisfying for traditional programming languages, it is not adequate for programming with graph transformation rules.

It may be argued that rule-based programming on graphs makes no sense for the example of sets. This is not true. What would a set specification with PROGRES look like, if we do not intend to realize sets by trees? In that case, we would choose another set structure, in which the fact that a set contains an element is expressed by an edge of type SetElem from set to element node explicitely. In fact, we think that the semantics of an abstract graph type SETS is expressed adequately by rules which add and remove elements by creating and deleting corresponding set membership edges. Figure 8 presents a graphical version of the transformation SetInsertValue. It makes use of negative context conditions. The crossed out graph elements in the rule must not find a match in the current working graph. In our example this means that the set given as parameter S may neither already contain the element given as parameter Elem, nor any other element with the same value attribute as Elem. In that case the rule matches and inserts Elem into S.

Of course, we do not want to replace the tree implementation of sets and their efficient access operations. But assuming that the client of the AGT SETS does not look into the internal body implementation when using the interface, we can pretend another realization. Figure 8 presents an interface, which "realizes" the transformation SetInsertValue graphically. Using the interface, the client does not notice whether the graph type is actually realized by another implementation or not. This is exactly the intention of data abstraction that implementations are interchangable provided they have the same interface. In fact, it does not matter whether intuitive rule-based or efficient imperative programming is used, the interface of SETS operations needs not to be changed. But, of course, since interface operations use the newly introduced edge type SetElem, this edge type must be declared as a part of the interface, too. Note that rules in the interface illustrate

```
interface SETS
  from ELEMENTS import ELEM, GetElemValue;
  node type SET end;
  node type ELEM end;
  edge type SetElem : SET [0:1] -> ELEM [0:n];
  transformation SetInsertValue( S    : SET ;
                                 Elem : ELEM) =
```




```
    condition GetElemValue ( ‘2 ) = GetElemValue ( ‘3 );
  end;
  . . .
end interface;
```

Figure 8: Graphical set manipulation

the effect of the corresponding transformations in the body and do not have any relevance for the actual execution of the corresponding operation. That means, they merely serve as “formal comments” with precise semantics in contrast (or in addition) to natural language. We have to admit that the interface transformations are, in general, not able to reflect the real effect of the implementation in the body. For example, it is not possible to deal with complex operations handling different special cases in a single rule. Nevertheless, whenever possible, the description of the essential effect of an exported graph transformation is expressed by a rule more accurately than by misleading comments.

As already mentioned, the client of an interface is not affected, whether an exported resource is realized by a single rule or by a complex tranformation, they are made available by their signature only. The internal structure of the underlying graph is not accessible and the implementation of graph transformations is not visible. But, is strict encapsulation of implementation details like the tree structure in red/black trees really what we intend to achieve with our graph transformation approach? Evidently, we have abstracted from the internal realization with graphs completely by offering interfaces which can be used in a procedural way for both, the `SETS` and the `RBTREES`. Consequently, implementations based on these interfaces can only be complex transformations using the imports as procedure calls connected by control structures. But programming with graph rewriting is motivated by the idea of modeling data structures as graphs and operations with graph rewrite rules.

As we have seen with the graphical "comment" interface in figure 8, this is even true for simple data structures like sets.

4. Specification–in–the–Large with Updatable Views

The trade-off between data abstraction and encapsulation on one hand and describing graph transformations by giving rules, which access nodes and edges directly, on the other hand is under permanent investigation. Different approaches have been suggested [26, 27], where specifications with different views on common underlying data are coupled, thereby allowing mixed access via procedural transformation calls and direct manipulation of abstract graph view elements.

Taking the specifications of sets so far, we already have both approaches available, the efficient implementation realized in the `SETS` body which has a procedural flavor and the graphical rule-based solution, which we have sketched for the `SETS` interface in figure 8. Until now, we have not undertaken any efforts to bring these solutions together. We have considered the interface rules as formal comments only. In fact, note that by introducing the edge type `SetElem` we have revealed the schema for a graph-based set realization in the interface, although the implementation never creates edges of this type. But, we can not prevent that some client imports this edge type which enables him to create instances of this edge type. In fact, this would even be the preferred style of visually manipulating sets by means of graph rewrite rules in `SETS` module clients.

The insertion rule for sets shown in figure 8 is very simple and does probably not represent a rule which is formulated by a user. This is mainly the case, because the user formulates graph transformations in a different, application specific context, where he wants to use the underlying graph structure on a higher level. Figure 9 depicts the situation that two sets are merged together in a union set. This is a typical operation, which is not already provided by `SETS` itself. The rule takes two set nodes given as parameters `S1` and `S2`, resp. and merges them by inserting all elements of the set `S2` into `S1`. The dashed double boxes represent set nodes of the language PROGRES, i. e. these nodes are able to match all the nodes in the current working graph that can be reached by the corresponding `SetElem` edges starting at `S1` or `S2`. This rule is much more comprehensible than the alternative imperative realization on top of the `SETS` interface that takes one element after the other from one set and inserts it into another set. Obviously, in the case, where the user creates set member edges explicitly, these edges have to be present in the graph afterwards and remain accessible. They will be checked for existence, they will be traversed to check properties of the element nodes, and they will be removed later on. Therefore, independent from the implementation specific underlying graph structure shown in figure 5 the client specification has a different *view* on the structure outlined in figure 10. In this figure heavy boxes and edges depict those graph elements which are visible for clients using the `SETS` interface view. The other elements in the figure are hidden for the client. Note that nodes of type `ELEM` are accessible as set elements without knowing that they are stored attached to internal tree nodes of

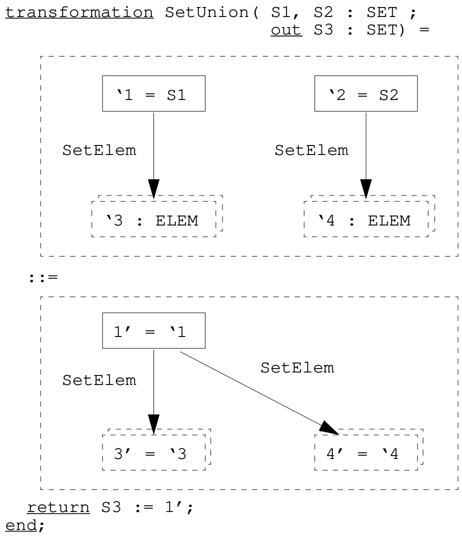


Figure 9: `SETS` client operation building the union of two sets

type `RBNODE`. Based on this view elements may be inserted and deleted by creating and removing edges. But, of course, we still do not want to remove the internal tree implementation allowing efficient queries instead of searching through all elements in the set.

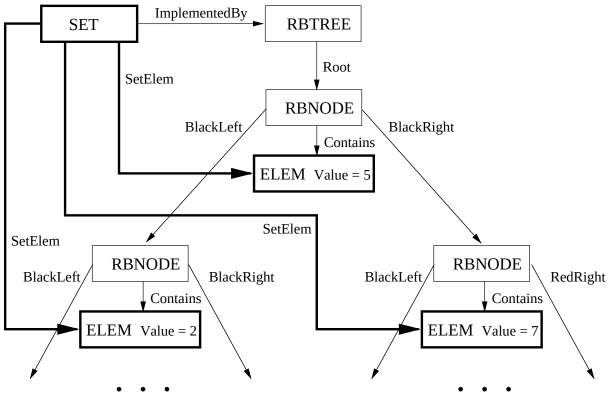


Figure 10: View on the common underlying graph structure

How can we manage to keep the graph, that we operate on, in a consistent state? This is no problem as long as we can be sure that only interface operations are invoked to perform graph modifications. But it is evident that operations which circumvent the set interface operations by manipulating exported node and edge type instances directly may result in inconsistent states. Two examples illustrate, what we mean be inconsistency. First, how can we ensure, that whenever an `SetElem` edge is created to indicate the membership of an element to the set, this element also will eventually be inserted into the red/black tree allowing its retrieval via the set interface? For this case we need to establish a coupling mechanism between two abstract graph types. But, we also have inconsistency with respect to the behavior of only one involved abstract graph type. For example, it is not allowed that two elements with the same value attribute are present in the set at the same time. How can we ensure that elements are not inserted twice by giving the user complete control over the internal graph structure? If we are not able to find answers to these questions, we can either forget the idea of data abstraction for graph rewriting completely or we have to dictate the use of interface operations for any access to the graph structure.

While the latter of the two sketched questions is concerned with the semantics of an abstract graph type, the former has a closer relation to the *view update problem* known from databases. That means, as long as the graphical `SETS` view is needed for abstracting from implementation internals in a passive way exclusively, we can realize it in the body of `SETS` by updating two almost separate graph structures, the set and the tree structure, at the same time. But, as soon as the set elements of the view are manipulated directly, we have to recognize the intention of that action and to find an appropriate operation on the underlying graph structure. Either, in the case of deletion and insertion, in order to keep the overall structure of sets and trees in a consistent state or, even in the case of a query, to benefit from the efficient tree implementation.

In the following we will suggest a solution for programming visually with abstract graph types, which keeps the underlying graph structures in a consistent state (with respect to different involved views). This comprises both, the internal semantics of a single abstract graph type and the consistency with respect to different coupled graph types, whenever the graph has been modified. Regarding our example we will ensure that our sets will not contain elements more than once and that all elements inserted into sets directly are also inserted into their underlying trees.

For the realization we have adopted concepts from the area of active database systems by introducing *active constraints* [38]. The object-oriented structure of data in modern database systems is comparable to the graph structure in PROGRES. The same holds for the intention to keep data in a consistent state. Event mechanisms as a special data integration strategy, so-called ECA (event-condition-action) rules, have been introduced for active database systems to perform this task. These rules react to events raised by modifications on data. After checking consistency conditions actions may be called that serve to repair the damages.

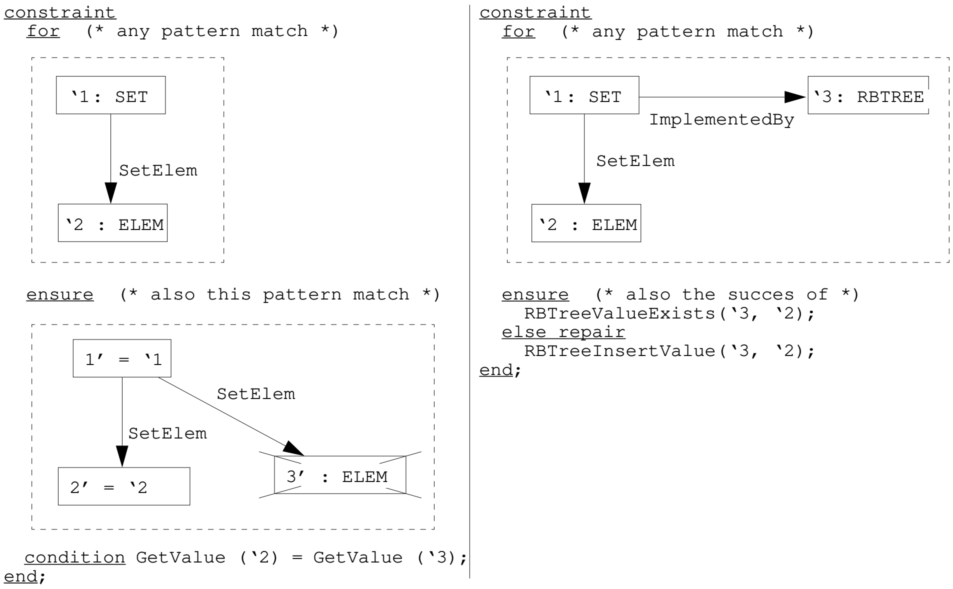
With this inspiration we have decided to introduce constraints in the language PROGRES. Besides the already mentioned *schema constraints* in figure 1 we are also able to formulate *integrity constraints* which follow in their expressive power the suggestion in [18]. They can be specified in textual or graphical form and consist essentially of two parts, the `for`-part and the `ensure`-part. Their semantics is that all matches of the `for`-part in the working graph must be extensible to matches of the `ensure`-part, too. If this is not possible, constraints — in their pure form — stop the execution. In the textual variant the graph patterns may be replaced by complex queries. In addition to the check whether a forbidden pattern in a graph exists, we have equipped the constraints with an *active character* similar to the repair actions in ECA rules. This allows to perform transformations after which a consistent state of the working graph is recovered.

These constraints serve the purpose of ensuring the consistency of the graph structure manipulated from different perspectives. As already mentioned, we want to apply constraints in two different ways. First, they have to ensure the semantics of single AGTs. The left constraint in figure 11 ensures that whatever the user does, there will never be two elements present in the set with the same key value. This is done by checking for every `SET/ELEM` pair the nonexistence of another set element with the same value as the value of `ELEM`. Since this constraint contributes to the observable semantics of the abstract graph type `SETS` it belongs to its interface. In addition, constraints are used for tracing set operations in the tree structure by invoking appropriate interface operations shown on the right side of figure 11. Here, we even expect the `ensure`-part to fail for the pattern of the `for`-part at least once. That means, an element was inserted into the set, but has not yet been inserted into the tree. In that case, we call the corresponding tree operation with the `else repair`-part explicitely. In that way constraints are purely a means for translating updates of the externally visible graph part (view) into necessary updates of the underlying hidden graph structure.

Interface constraints have no repair action because they define the invariants of the regarded graph type and not its implementation. Therefore, they contribute to the observable behaviour and should not reveal internal realization details. Besides these actions would only be able to invoke interface operations. But, in practice, interface operations are not appropriate as repair actions because they assume to manipulate proper and consistent structures. Other means are necessary to ensure that duplicates are either avoided or, at least, detected and removed. This can be done either by testing whether a value is present in the set before it is inserted or by providing a corresponding body constraint which removes duplicate value elements from the set. If no measures are taken with this respect and the interface constraint is violated the execution will stop. Note that the constraint resembles the graphical insertion operation in the interface with the left and right side exchanged.

The body constraint in figure 11b) mixes graphical and textual representation. This is due to the fact that we check for graph patterns created by an analogous graphical rule. But, since our `RBTREES` interface was created without the ambition to apply rules on the internal structure, we are only able to invoke the appropriate interface operation in a procedural way for both, checking the existence of the respective element in the tree and for inserting it, if necessary.

To summarize, the construction of updatable graph views resovles the contradiction between full data abstraction and visual programming with graph rewrite rules



a) interface constraint defining invariant of AGT

b) body constraint solving the view update problem

Figure 11: `SETS` module with integrity and (view) update constraint

in principle. But there are still many *open questions*, which have to be answered before the presented approach should be used extensively:

- Is it always possible to determine automatically, when constraints and their associated repair action have to be checked and triggered?
- What about the efficiency of the currently implemented constraint checking algorithm for those subclasses of (active) constraints used in practice?
- Is the extensive usage of active constraints (ECA-rules) with all well-known problems concerning the interaction of concurrently triggered constraint violations (events) and the termination of repair actions triggering further constraint violations etc. too dangerous from a software engineer's point of view?
- Is it always feasible to develop a consistent set of active constraints which are able to translate all updates of a nontrivial view into updates of the underlying hidden graph structure?

5. Conclusion and Future Work

The language PROGRES and its tools are the result of many years of application-oriented graph grammar research activities. Nowadays, they are used at various sites (like Carleton University - Ottawa, INRIA - Sophia Antopolis, University of

Antwerp, and many other places) for specifying and prototyping software, database, and knowledge engineering tools. Despite our success in demonstrating the usefulness of graph grammar engineering concepts and tools for software development, we still have to solve many problems, until they are really useful for medium to large-scale industrial projects. Compared with the history of programming languages, PROGRES has now reached the state of languages like Algol-68 or Pascal. It has a sophisticated type system and enforces a well-structured style of programming. But its current release (available via the PROGRES home page [b]) gives not yet any support for specification–in–the–large activities.

Considering the pile of already published module concept papers in the last 30 years, it was and is clearly not our goal to start from scratch and to (re-)invent concepts for programming–in–the–large activities. On the contrary, we were starting with a well established module concept as it is part of Modula-2 or Ada. As a consequence, we have only one type of modules and one type of (import/use) relationship between them. Modules in this sense are just building *visibility boundaries* around arbitrary sets of type and operation declarations.

This decision does not only simplify the incorporation of modules into the PROGRES system, but gives its users the flexibility to adhere to quite different popular styles of programming–in–the–large:

- Modules with type declarations only in their interfaces and with empty bodies are a means to decompose complex graph (database) schemata in the same way as it is done in database systems like PCTE [13].
- Modules which export a single transformation (with optional pre- and post-conditions) allow the definition of graph transformation units as suggested in [20].
- Modules which summarize and propagate the exported resources of a number of imported modules to their own interfaces, offer more or less the concept of horizontally structuring graph data types as presented in [17].
- Modules which export a single "main" node type and a number of related operations at their interfaces are our means for defining abstract data types in the classical sense [29].

All these different programming–in–the–large styles have their own advantages and disadvantages. Even the most important software engineering principle, building abstract data types with hidden implementation details, has its severe drawbacks when it is combined with the idea of programming with graph rewrite rules. We have seen that any module built on top of an abstract graph type module in the classical sense has no longer access to the nodes and edges of the underlying graph data structure. As a consequence, all graph transformations have to be programmed by calling operations of the imported AGT module only, i. e. the idea of programming

[b] http://www-i3.informatik.rwth-aachen.de/research/progres/

with graph rewrite rules instead of constructing sequences of procedure calls is no longer applicable.

Our solution for resolving the conflict between the principle of data abstraction and the paradigm of visual programming with graph rewrite rules was based on the construction of graph views. Modules which reveal part of their node and edge types as well as attribute declarations at their interfaces are used for this purpose. Their clients are allowed to access instances of exported node and edge types directly within their own graph rewrite rules. We have seen that the direct manipulation of exported type instances in rewrite rules of client modules may result in inconsistencies of the graph structure as it is implemented by the imported server module. The offered solution for our version of the well-known update view problem is based on the idea of active database systems to use ECA-rules for this purpose. Constraints with associated repair actions are our means to translate the insertion or deletion of exported type instances into necessary modifications of the hidden graph structure. Further details concerning supported types of constraints and the implemented, incrementally working constraint checking algorithm are the subject of another forthcoming publication.

Finally, we have to emphasize that the presented graph grammar module concept offers (hopefully) valuable means for the development of large PROGRES specifications, but gives no proper support for

- aggregation of subgraphs to complex node objects with associated graph transformation methods,
- (vertical) refinement of graph types and their operations,
- and modeling of distributed concurrently (inter-) acting graph objects,

First results of ongoing research activities concerning these problems may be found in [12, 17, 36]. Major efforts are still necessary to refine and merge their proposals for hierarchical graph data models, inheritance or refinement of graph types, and distributed programming with graph rewriting systems. It is a longer term research goal of the GRACE initiative to develop a single generic graph- and rule-based language and environment which offers all these concepts and which may even be instantiated with rather different types of graph data models and rewrite rules.

References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph Transformation for Specification and Programming. Technical Report 7, University of Bremen, 1996.
2. C. Beierle. Concepts, Implementation, and Applications of a Typed Logic Programming Language. In Beierle and Plümer [3], chapter 5.
3. C. Beierle and L. Plümer, editors. *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*. Elsevier Science B.V., 1995.

4. J. Börstler. *Programming in the Large: Languages, Tools, Reuse*. Dissertation, RWTH Aachen, 1993. (in German).
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1992.
6. A. Corradini and U. Montanari, editors. *Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA '95)*, volume 2 of *Electronical Notes in Theoretical Computer Science*. Elsevier Science B.V., 1995.
7. A. Cremers, T. Lemke, and M. Rohen. Logic Program Modules for Interoperable Information Systems. In Beierle and Plümer [3], chapter 4.
8. J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors. *Proc.* 5^{th} *Int. Workshop on Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
9. F. DeRemer and H. Kron. Programming–in–the–large versus Programming–in–the small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.
10. H. Ehrig and G. Engels. Pragmatic and Semantic Aspects of a Module Concept for Graph Transformation Systems. In Cuny et al. [8], pages 137–154.
11. G. Engels, C. Lewerentz, and W. Schäfer. Graph grammar engineering: A software specification method. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proc.* 3^{rd} *Int. Workshop on Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 186–201. Springer-Verlag, Berlin, 1987.
12. G. Engels and A. Schürr. Encapsulated Hierarchical Graphs, Graph Types, and Meta Types. In Corradini and Montanari [6].
13. European Computer Manufacturers Association (ECMA). ECMA Standard 149: Portable Common Tool Environment (PCTE) Abstract Specification, 1990.
14. E. Feigenbaum. On Generality and Problem Solving: A Case Study Using the DENDRAL Program. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 6. Edinburgh University Press, 1971.
15. H. Göttler, J. Günther, and G. Nieskens. Use Graph Grammars to Design CAD-Systems! In H. Ehrig and H.-J. Kreowski, editors, *Proc.* 4^{th} *Int. Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 396–410. Springer-Verlag, Berlin, 1990.
16. V. Haarslev, editor. *Proc.* 11^{th} *Int. IEEE Symp. on Visual Languages (VL '95)*, Darmstadt, Germany, Sept. 1995. IEEE Computer Society Press, Los Alamitos, CA.
17. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and Vertical Structuring in Typed Graph Transformation Systems. *Mathematic Structures in Computer Science*, 6(6):613–648, December 1996.
18. R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars. In Corradini and Montanari [6].
19. M. Himsolt. GraphEd: An Interactive Graph Editor. In *Proc. STACS 89*, volume 349 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989.
20. H.-J. Kreowski and S. Kuske. On the Interleaving Semantics of Transformation Units — A Step into GRACE. In Cuny et al. [8], pages 89–106.
21. M. Lefering, M. Nagl, A. Schürr, B. Westfechtel, A. Winter, and A. Zündorf. Internal Conceptual Modeling: Graph Grammar Specifications. In Nagl [25], chapter 3.
22. M. Löwe and M. Beyer. An Implementation of Algebraic Graph Rewriting. In *Proc.* 5^{th} *Int. Conf. on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 451–456. Springer-Verlag, Berlin, 1993.
23. M. Minas and G. Viehstaedt. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In Haarslev [16], pages 203–210.
24. M. Nagl. *Software Engineering: Methodological Programming in the Large*. Springer-

Verlag, Berlin, 1990. (in German).
25. M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1996.
26. M. Nagl and A. Schürr. Software Integration Problems and Coupling of Graph Grammar Specifications. In Cuny et al. [8], pages 155–169.
27. M. Nagl, A. Schürr, and A. Winter. Specification–in–the–Large for Graph Grammars. In Nagl [25], chapter 5.2.
28. A. Newell and H. Simon. GPS, a Program that Simulates Human Thought. *CT*, pages 279–293, 1966.
29. D. Parnas. A Technique for Software Module Specifications with Examples. In *Comm. of the ACM*, volume 15, pages 330–336. ACM Press, 1972.
30. P. Robinson. *Hierarchical Object-Oriented Design*. Prentice Hall, Engelwood Cliffs, NJ, 1992.
31. A. Schürr. *Operational Specification with PROgrammed Graph REwriting Systems*. Deutscher Universitäts-Verlag, Wiesbaden, 1991. (in German).
32. A. Schürr. Rapid Programming with Graph Rewrite Rules. In *Proc. USENIX Symposium on Very High Level Languages*, pages 83–100. USENIX Association, 1994.
33. A. Schürr. Programmed Graph Transformations and Graph Transformation Units in GRACE. In Cuny et al. [8], pages 122–136.
34. A. Schürr, A. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer and P. Botella, editors, *Proc. 5^{th} European Software Engineering Conference (ESEC '95)*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, Sitges, Spain, Sept. 1995. Springer-Verlag, Berlin.
35. A. Schürr, A. Winter, and A. Zündorf. Visual Programming with Graph Rewriting Systems. In Haarslev [16], pages 326–335.
36. G. Taentzer and A. Schürr. DIEGO, another step towards a module concept for graph transformation systems. In Corradini and Montanari [6].
37. P. Wegner. *Programming with Ada*. Prentice Hall, Engelwood Cliffs, NJ, 1980.
38. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, 1996.
39. R. Wiener and R. Sincovec. *Software Engineering with Modula-2 and Ada*. Wiley & Sons, New York, NY, 1984.
40. N. Wirth. *Programming with Modula-2*. Springer-Verlag, Berlin, 1985.
41. A. Zündorf. *A Development Environment for PROgrammed Graph REwriting Systems*. Deutscher Universitäts-Verlag, Wiesbaden, 1996. (in German).