Chapter 1

Introduction

1.1 Visual Languages and Graph Transformation

The origin of visual languages, or more precisely, of executable visual programming languages lies about 30 years in the past. At the beginning, research on visual programming was related to the main topic of this tutorial: graph grammars or graph transformation (rewriting) systems. Some of the earliest visual programming languages like AMBIT/G [Chr68] or AMBIT/L [Chr68] of Carlos Christensen and especially PLAN2D [DFS74] of Denert, Franck, and Streng are indeed graph transformation languages. Furthermore, Lindenmeyer systems [RS74], which are closely related to graph grammars, were and are still used for modeling the growth of plants and for picture generation and animation purposes [RS86].

Later on, the graph transformation and the visual language research areas developed more or less independently from each other until the end of the eighties. Visual programming research had a main emphasis on designing and implementing programming languages and environments [VL86, ... , VL97], whereas most graph grammar researchers spent their time with the formal definition and comparison of various types of graph grammars [IWGG]. Since the end of the eighties, people are starting to cross the borderlines between the two communities. On one hand, graph grammars are nowadays used for the definition of the syntax and semantics of visual languages and for generating syntax-directed editors and parsers from these specifications. Examples of this kind may be found in [MV95, RS97, ZZ97]. On the other hand, there is a growing number of visual programming languages and environments, which rely directly on the graph transformation paradigm. Some examples of this category are the programmed graph rewriting languages PAGG [GGN91] and PROGRES [SWZ95a] as well as the visual database programming language GOOD [PBA+92] and the graph manipulation language $Ludwig_2$ [Pfe95].

To summarize, the relationships between visual (programming) languages and graph transformations are as follows:

- Graph transformation based languages are a special brand of rule-based visual programming languages with precisely defined syntax and semantics.
- Various types of graph grammars are used to specify the syntax (and sometimes also the static semantics) of visual languages.
- Finally, graph transformation based meta programming tools support the development of complete visual programming environments (with syntax-directed editors, parsers, animation tools etc.).

These notes as well as the tutorial itself have the main emphasis on visual programming with graph transformations (cf. chapters 3 and 4). The related topics of modeling visual languages and generating visual language environments are briefly addressed in chapters 5 and 6. For the closely related topic of parsing visual languages based on graph grammars the reader is referred to [RS97].

1.1.1 History of Graph Transformations

Graph grammars and graph transformation systems were invented about 25 years ago as so-called "web grammars" [Sch70] and "Chomsky systems for partial orders" [PR69]. From the very beginning graph grammars and graph transformation systems have been used to solve real world problems in the field of computer science, biology, and so forth [CER79]. In the beginning pure graph transformation systems with rather simple forms of rules were used. But soon it became obvious that additional concepts were needed to make the description of complex software engineering tools feasible. One of the most important extensions was the introduction of new means for controlling the application of graph rewrite rules. This lead to the definition of PROgrammed Graph REwriting Systems, with PROGRES being one example of this kind. Many intermediate steps were necessary before the visual graph transformation programming language PROGRES evolved into the form documented in chapter 3 of these notes.

In the meantime Herbert Göttler developed rather similar graph transformation approaches, so-called Y and X graph grammars [Göt83, Göt88]. Both Y/X graph grammars and all predecessors of PROGRES use *set theory* to define the semantics of their graph transformation rules. Presented definitions state precisely under what conditions a graph transformation rule's left-hand side matches a subgraph of the regarded graph and how the rule rewrites the graph by replacing the matched subgraph with a copy of its right-hand side. The set-theoretical approach is also called the *algorithmic* approach due to the constructive flavour of its definitions. It has a surprisingly large number of variants, which differ mainly with respect to the underlying data model (e.g. directed graphs versus hypergraphs), the form of the left-hand sides of rules (e.g. a single node or a whole graph), the way how a copy of a rule's right-hand side is embedded in the preserved part of the rewritten graph, and the available means for controlling the application of rules. The most comprehensive survey of algorithmic graph rewriting approaches based on a data model of directed graphs is [Nag79a], the most comprehensive monography for the data model of hypergraphs is probably [Hab92].

There are two other main branches of graph transformation systems, beside the algorithmic approach with its set-theoretic fundament. The more popular one uses category theory as the underlying formalism, the other one is based on predicate logic. The category-theoretical approach was developed in the early seventies at the Technical University of Berlin [EPS73]. It is called the *algebraic* graph transformation approach for the following reasons: The application of a rule to a given graph is defined by an algebraic construction, the so-called pushout diagram of category theory [Ehr87]. Its rules are less expressive concerning the treatment of context edges of to be deleted nodes, but more suitable for the definition of parallel graph transformation approaches, a subject which is out of the scope of the tutorial. For further details concerning this topic the reader is referred to [CMR+97].

The *predicate logic* graph transformation approach exists in two rather distinct forms. The first one, developed by Courcelle, is based on monadic second order theory and related to the algebraic graph transformation approach [Cou97]. It uses graph grammars for generating graph languages and predicate logic formulas for the definition of decidable graph-theoretic properties for generated graph languages. The other logic-based approach uses first order predicate logic only [Sch97b]. It is closely related to the set-theoretical graph rewriting approach and was developed as the underlying formalism of the language PROGRES.

Nowadays the graph transformation paradigm has reached a certain degreee of maturity. Graph transformation languages(graph grammars) are known or used by a growing number of people in rather different communities as shown in the following chapters. For further details concerning the foundations and applications of graph transformations in various fields inside and outside the field of computer science the reader is referred to the already published first and the forthcoming second volume of the "Handbook on Graph Grammars and Computing by Graph Transformation" [Roz97, Roz99]. Finally, we should mention some serious efforts to develop a common framework of graph transformation concepts and tools. The project GRACE — for Graph and Rule Centered Environment — has for instance the long-term goal to offer potential users of graph transformation languages not one hundred different approaches with undocumented (dis-)advantages for certain application areas, but one generic adaptable approach instead [AEH+96]. The project GRACE as well as various other activities related to the disemination of graph transformation knowledge were and are still sponsored by the European Community as part of the two working groups COMPUGRAPH (Computing by Graph Transformation) and GETGRATS (General Theory of Graph Transformation Systems) as well as through the TMR network APPLIGRAPH (Applications of Graph Transformation).

Chapter 2

Notations and Mechanisms for Graph Productions

Chapter 3

The Graph Transformation Language PROGRES [1]

3.1 Object-Oriented Modeling and Graph Transformation

It is one of the main software engineering research tasks to develop languages, tools, and methods for the construction of increasingly complex software systems for application areas like computer integrated manufactoring, chemical engineering, office automation, project management, hypertext editing, and computer-aided software engineering itself. Software systems for these application areas have the common characteristics to store, retrieve, modify, and display complex structured, distributed, but nevertheless to be integrated sources of information.

The systematic design and realization of appropriate internal data models for these systems and their accompanying data access operations is a challenging task. It requires languages, tools, and methods for

- the visual design of appropriate graph-like data models, which take objects as well as relationships between objects into account,
- the high-level visual description of needed queries and update operations on related sets of objects (graph transformations),
- the validation of once constructed data models and their operation descriptions,
- and the translation of these descriptions in correct and efficiently working as well as portable and extendible system implementations.

Object-oriented modeling languages and their accompanying CASE tools are nowadays probably the most popular means for these purposes. They offer class diagrams for the design of graph-like data models, state transition diagrams for specifying the behavior of single objects, and object interaction diagrams (message sequence charts) for describing the exchange of messages between different objects.

One of the main drawbacks of the object-oriented modeling paradigm as implemented by the new OMG standard UML [Rat97, FS97], the Unified Modeling Language, and its predecessors developed by Booch, Rumbaugh, and Jacobson may be characterized as follows: Almost all types of diagrams, which are used for modeling the behavior of objects, rely on the notion of methods, i.e. on operations, which are attached to a single object (class). This approach is appropriate as soon as the concrete architecture of a software system has to be designed, but causes some problems during software analysis or early software design phases. In the latter case, one should not be forced to associate regarded global system operations with a often more or less arbitrarily chosen object class.

The activity diagrams of UML are a first attempt to offer "new" visual means for the definition of bussiness or software system processes on a higher level. They describe required or permitted sequences of activities

[1] Chapter 3 is an excerpt from the PROGRES chapter in [Roz99].

without translating them into methods of single objects. These activity diagrams belong to the class of visual (programming) languages which model the flow of control of a regarded system. They do not offer any (visual) means for describing concrete operations that manipulate sets of related objects.

The object-oriented modeling language (method) Fusion [CAB94] attacks the discussed problem in a different way by offering so-called *system operation schemata*. These schemata describe the state of a regarded subset of objects and their relationships (a subgraph of the whole object graph) before and after the execution of a system operation with pre- and postconditions. It uses so-called *life cycle expressions* for modeling permitted or required sequences of system operation calls. The main drawback of the Fusion approach is the lack of any visual notation for the definition of pre- and postconditions.

This is the moment where the graph transformation approach comes into play. It offers a visual and precisely defined notation for the definition of global system operations, which manipulate sets of related objects (object graphs). A visual language, which combines the object-oriented modeling approach of Fusion with the graph transformation paradigm should have about the following elements:

- Class diagrams for the object-oriented design of data models.
- Graph transformation rules for the specification of executable system operations.
- Control structures (life cycle expressions) for "programming" complex system operations

The language PROGRES and its programming environment have been designed having these requirements in mind. They are based on the data model of directed attributed graphs and offer the concept of *PROgrammed Graph REwriting Systems* for the description of complex graph transformations. The language combines rule-based, object-oriented, deductive and active database system as well as imperative programming concepts for this purpose. Its semantics is formally defined based on the rather general formalism of programmed graph or structure replacement systems [Sch97b].

PROGRES is a *hybrid visual language* with a mixture of tightly integrated graphical and textual elements: a constructed graph transformation specification is a running text that contains graphical elements for the left- and right-hand sides of graph transformation rules, which contain in turn text blocks for the definition of additional application conditions, and so forth. Furthermore, any language construct with a graphical representation has a text representation, too. The latter one may be manipulated using plain text editors.

The accompanying programming environment supports syntax-directed editing of graphical and textual language elements, incremental parsing of manipulated text representations, analysis of several hundreds of type checking rules, and execution of constructed graph transformations. It was originally designed for specifying and generating (components of) tightly integrated software engineering tools as explained in [Nag96]. It is nowadays used at various sites around the world, mainly for specifying and rapid prototyping interactive tools with graphical user interfaces:

- Our colleagues in Aachen need PROGRES for rapid prototyping configuration management and process modeling tools [Wes96] as well as for constructing software design and reverse engineering tools [Cre98].
- Software engineers at the University of Leiden have been using PROGRES for the specification of visual (database query) languages and for prototyping process modeling tools [AE94, Zam96].
- There are also activities at INRIA, Sophia Antopolis, to manipulate Conceptual Graphs [Lap96] and at Carleton University to develop distributed system analysis algorithms by means of PROGRES [Hri98].

The following section 3.2 starts the presentation of PROGRES with a discussion of its relationships to well-known executable specification, visual programming, and rapid prototyping languages. Section 3.3 introduces then all language constructs needed for the definition of graph schemata, whereas section 3.4 deals with all those language constructs needed for the declaration of graph queries and transformations. Section 3.5 changes the focus from "Specification-in-the-Small" activities to "Specification-in-the-Large" activities, i.e. it introduces a graph transformation module concept that it is not yet an implemented part of the PROGRES programming environment. The latter one is the main subject of section 3.6. Its intention is to provide the interested reader with an impression of the "look and feel" of all programming environment components as well as of the generated prototypes.

3.2 Related Specification and Programming Languages

Writing a compact comparison of PROGRES with related work is a rather challenging job. There are too many aspects which have to be taken into account due to the fact that PROGRES is an object-oriented data modeling language, a formal specification language, a visual programming language, a meta programming language for generating software engineering tools, and one example of a graph transformation language. The following subsections are nevertheless an attempt to cover all these aspects.

3.2.1 Semiformal Modeling and Formal Specification Languages

Traditionally used structured analysis and design methods — as for instance those proposed by Hatley and Pirbai [HP87] or Yourdon [You89] — as well as more recently developed object-oriented analysis and design methods — like those proposed by the "Three Amigos" Booch [Boo94], Jacobson [Jac94], and Rumbaugh [RBEL91] or those proposed by Coleman et al. [CAB94] — suffer from the fact that they do not possess a precisely defined syntax and semantics. They are very useful for describing the requirements and the architecture of a software system on a semiformal level. But these descriptions cannot be used for verifying certain properties of the system under construction.

Formal specification languages — as for example CIP-L [Gro85], Larch [GH93], or Z [Dil92] — on the other hand possess a rigorously defined syntax and semantics. They support the verification of modeled system properties or rely on property perserving refinement of abstract specifications into concrete implementations. But their underlying formalisms of heterogeneous algebras, (typed) predicate calculi, or set theory are not tailored towards the definition of graph-like data structures and the specification of graph transformations. They rely on more general data models (like relations) and rather low-level data modifying operations (like insert tuple into relation), on top of which the desired graph data model together with a graph pattern matching and transformation mechanism has to be "implemented". As a consequence, serious efforts are under way to combine the graphical notations of object-oriented modeling languages for data and operation modeling purposes with subsets of formal specification languages like Z. First results of this kind are Syntropy [CD94], a combination of Z and Rumbaugh's method OMT, as well as Catalysis [DW96], a combination of Z and Coleman's method Fusion.

The currently developed *Unified Modeling Language* (UML) [FS97] follows the same trait of thoughts by offering a so-called Object Constraint Language (OCL) for the definition of derived attributes and relationships, static integrity constraints for certain object classes and associations, or pre- and postconditions of methods. OCL still suffers from the fact that it has neither a well-defined underlying type concept nor a precise semantics definition, comparable to the type and semantics definition for attribute and path expressions in [Sch91]. Furthermore, neither UML nor its predecessors resolve the contradiction between rather high-level graphical notations for designing data structures and rather low-level text-oriented means for defining the effects of single object-modifying methods. A first proposal for specifying object behaviors based on a graphical notation is published in [KHCM98]. The authors of this paper suggest the usage of snapshots (object diagrams) as pre- and postconditions of operations. Such a pair of pre- and post-snapshots comes very close to the definition of a graph transformation rule, without having its precisely defined semantics.

3.2.2 Visual Rule-Based Programming Languages

Changing our focus from specification or modeling activities to lower-level programming activities we have to regard that category of visual programming languages which contains all graph transformation languages as a special subclass. These are the rule-based visual programming languages.

Disregarding early graph transformation languages such as AMBIT/G or PLAN2D our history of *visual rule-based programming languages* starts some years later on with systems like BITPICT [Fur91], ChemTrains [BL93], KidSim [SCS94], Vampire [McI95], and PictorialJanus [KS90]. All of them, except PictorialJanus, belong to the category of *icon rewriting languages*. Their underlying knowledge base is not a set of facts with a superimposed graphical representation, but a two-dimensional picture, usually called workspace or grid, with or without an underlying logical interpretation.

All these languages adhere to the simple *recognize-select-execute* model of rule-based languages. Despite their common execution model, visual rule-based languages differ from each other with respect to many aspects like the structure of their workspace, the details of the pattern matching process, and so on:

- In BITPICT [Fur91] workspaces as well as LHS and RHS of rules are simple pixel grids; a rule matches an area of the workspace which is identical to its LHS after geometric reflection and rotation operations. The system does not support modeling of higher-level entities.
- KidSim [SCS94] uses different types of icons instead of a single type of pixel as grid entries. Its matching process does not involve geometric reflection and rotation operations, but supports abstraction by means of subclassing. A more general icon in a rule's LHS matches any more specific icon on the workspace. Icons may possess properties (like size) and rules may query and manipulate these properties.
- Vampire [McI95] is another step further on from the simple data model of bit matrixes to logical graph structures. It adds the concept of abstract spatial relationships — like "above" — and logical connections between icons to the pure icon rewriting concept of KidSim. It is, therefore, no longer necessary to write n different rules for processing all pairs of icons which are 1 to n grid units above each other.
- ChemTrains [BL93], finally, is a kind of missing link between pure icon rewriting languages and graph rewriting languages, which are no longer sensitive to geometric relations of regarded objects. It manipulates graphical objects like boxes and lines and regards only two types of spatial relationships between them: connects and contains. It is therefore no longer well-suited for manipulating object representations.

Comparing PROGRES with those rule-based visual languages introduced above we have to state that languages like BITPICT [Fur91] or ChemTrains [BL93] focus on manipulation of data only. Even Vampire [McI95] with its class hierarchies and icon rewriting rules comes without a rigid type concept and without any type checking tools. Therefore, all these systems postpone recognition of programming errors to runtime. With respect to its graph type definition capabilities, PROGRES is more similar to visual database programming languages such as GOOD [PBA+92]. But these languages have less expressive pattern matching and replacing constructs. Furthermore, neither the above mentioned languages nor — to the best of our knowledge — any other visual rule-oriented language offers nondeterministically working control structures together with the ability to backtrack out of dead-ends of locally failing rule-application steps.

3.2.3 Graph Transformations and Meta Programming Tools

Changing our focus from VHL languages and visual programming languages to formalisms and meta programming tools, which are used for the construction of software engineering tools or compilers, we come across attribute grammars and tree transformation systems. Well-known tools in this area are CPSG [RT88] and PSG [BS86]. Rather recently a number of software engineering tool or compiler construction projects abandoned the rather restrictive data model of attributed trees used in these systems and adopted the graph-based approach of the IPSEN/PROGRES project [Nag96]:

- The hypergraph grammar based system DiaGen (presented in various stages of development at VL'93 through VL'98), generates complete visual programming environments.
- The system GenGEd is presented for the first time at VL'98 and built on top of the general purpose graph transformation system AGG. It is a visual editor generator which combines graph transformation with constraint solving technology for computing the layout of visual sentences.
- Optimix is a program optimizer generator, which relies on restricted classes of graph transformation systems for generating efficiently working program (data flow) analyzers and program transformation tools [Aßm96].

All these systems were designed having rather specific purposes in mind instead of being general purpose graph transformation languages such as PROGRES (or the just developed system AGG mentioned above). For further details concerning these graph transformation tools and their comparison with PROGRES the reader is referred to the forthcoming 2nd volume of the "Handbook on Graph Grammars and Computing by Graph Transformation" [Roz99] or the WWW-page [URL98].

3.3 Graph Schemata and Derived Graph Properties

This section introduces the basic elements for defining classes of graphs. It starts with the explanation of the chapter's running example in subsection 3.3.1, a modified version of the library information system (`LIS`) case study in [EP98]. Graph schemata — in the sense of conceptual database schemata — introduce all needed types of nodes and edges (cf. subsection 3.3.2) as well as their associated attributes (cf. subsection 3.3.5 and 3.3.6) and static integrity constraints (cf. subsection 3.3.6). The definition of appropriate attribute data types is outside the scope of PROGRES and delegated to its "host" programming language C (cf. subsection 3.3.4). PROGRES itself knows only a number of primitive data types and allows the definition of new functions over these data types and their access operations (cf. subsection 3.3.3).

3.3.1 The Running Example and DIANE Graphs

Modeling the data structures and operations of the `LIS` database is one of the standard case studies of the software engineering and the database management literature. Using the UML diagrams of [EP98] as a starting point, we will use the data structures and the operations of a library database as our running example, too.

First of all we have to clarify what kind of information has to be stored in a our library database and how this information has to be encoded using the data model of *DIrected, Attributed, Node and Edge labeled graphs*. These graphs are sometimes called DIANE graphs. They contain attributed labeled nodes instead of classified UML objects and directed labeled edges instead of binary UML associations between objects.

Node identifiers of DIANE graphs allow one to distinguish between nodes which have the same label and equal attribute values. There is no similar concept of edge identifiers, i.e. edges are treated as triples of the form (source node identifier, edge label, target node identifier). As a consequence, it is not possible to create two different edges between two nodes which have the same label and direction.

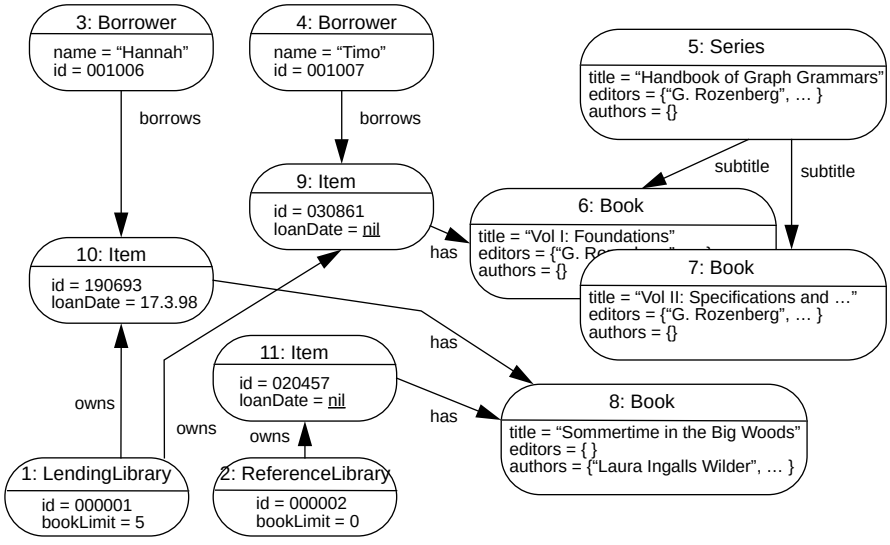


Figure 3.1: Manipulating precedence networks with graph transformations.

The graph of figure 3.1 distinguishes — in accordance with [EP98] — between published `Book` titles on one hand and available `Item` instances of these titles on the other hand. The general information about a certain book title consists of its `title` text, a possibly empty set of `authors`, and a possibly empty set of `editors`. Additionally needed attribute fields for publisher, ISBN number, and so forth have been omitted due to lack of space. Associated `Item` nodes are used to keep track of how many `Book` title instances are part of the library and their `Borrowers`. All `Item` nodes have a unique identifier `id` as well as an optional `loanDate` attribute. The later one has a well-defined date value, whenever the corresponding `Item` is lent out to a certain person. Otherwise, it has the undefined value `nil`.

The data model suggested in [EP98] has been extended such that we are able to deal with "part of" relationships between titles. One example of this kind is the displayed `subtitle` relationship between the "Handbook of Graph Grammar and Computing by Graph Transformation" `Series` and its first two volumes. Furthermore, we are using explicit `LendingLibrary` and `ReferenceLibrary` nodes. This allows us to store information about different (sub-)libraries in one common database.

In the following we will define the (graph) schema of our `LIS` database and some update and query operations for adding titles, borrowers etc. to the database as well as for locating book items, which have a certain keyword in their title.

3.3.2 Node Classes, Node Types, and Edge Types

PROGRES offers the following syntactic constructs for defining the intrinsic structural components of the corresponding class of `LIS` graphs and their legal combinations. These are:

1. *Node types* such as `Borrower` or `Book`, which are used as node labels and determine the static properties of their node instances.

2. *Edge types* such as `borrows` or `has`, which are used as edge labels and which impose some restrictions concerning legal edge source and target node types.

A situation that occurs very frequently when defining graph schemata in general is that many node types and corresponding edge type definitions become very similar. As an example, consider nodes of type `Book` or `Series` in figure 3.1. Nodes of these types possess `title`, `editors`, and `authors` attributes. Therefore, it was natural to introduce an additional concept which allows us to define common node type properties once and for all and to inherit them to node types as needed. These are so-called *node classes*. PROGRES enables us to build inheritance hierarchies for node classes. As usual in object-oriented languages, the "is a" notion is used for inheritance relationships. Multiple inheritance may be used to cut down the size of graph schema definitions considerably. This fact is shown in figure 3.2, which contains a graphical representation of our `LIS` graph schema. It has to be read as follows:

- Normal boxes represent node classes, which are connected to their superclasses by means of dashed edges representing "is a" relationships. `LIBRARY` is for example a subclass of `DB_OBJECT`. Node classes are *abstract classes* in the sense of object-oriented programming languages. They do not possess any direct node instances.

- Boxes with round corners represent node types, which are connected to their uniquely defined classes by means of dashed edges, too. The type `Series` belongs for example to the class `TITLE`. Node types correspond to non-abstract *final classes* of object-oriented programming languages. They possess direct node instances, but may not have any subclasses.

- Solid edges between node classes or node types represent *edge type* definitions. The edge type `owns` defines for example a binary relationship between `LIBRARY` and `Item` nodes, i.e. any source node of `owns` edges has to be an instance of `LendingLibrary` or `ReferenceLibrary`, any target node has to be an `Item` instance. Directed edges distinguish between source and target nodes, but may nevertheless be traversed in both directions.

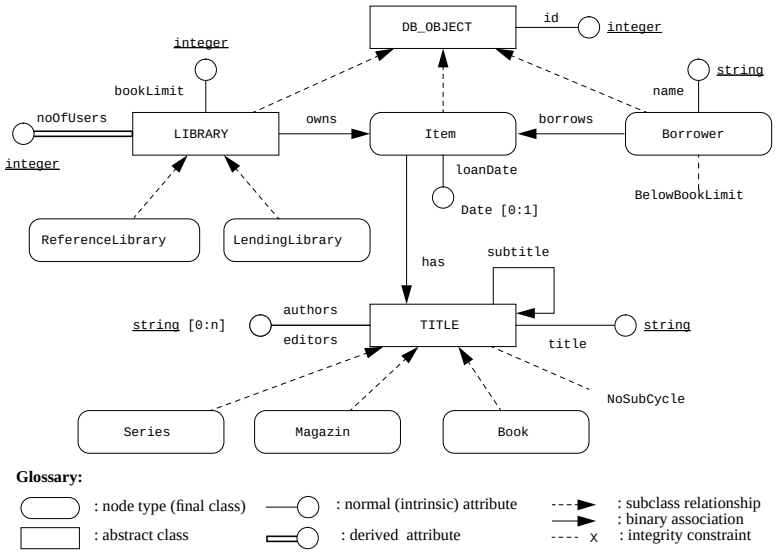


Figure 3.2: Graphical Representation of `LIS` Graph Schema.

The remaining elements of figure 3.2, which are used to declare node attributes and integrity constraints, will be introduced later on in this section and the following section 3.3.6.

To complete the explanation of language features for constructing graph schemata, we have to describe how *edge type cardinalities* may be defined. For this purpose we switch from the graphical schema definition notation of figure 3.2 to the more detailed textual representation of figure 3.3. The figure contains additional cardinality constraints for edges, which are not expressible using the graphical notation.

```
node class DB_OBJECT end;
node class LIBRARY is a DB_OBJECT end;
node type LendingLibrary : LIBRARY end;
  ...
edge type owns : LIBRARY [1:1] -> Item [0:n];
edge type has : Item [0:n] -> TITLE [1:1];
node type Borrower : DB_OBJECT end;
edge type borrows : Borrower [0:1] -> Item [0:n];
```

Figure 3.3: Textual representation of `LIS` graph schema.

Four different kinds of cardinality constraint qualifiers are available for the definition of edge types:

1. A qualifier `[0:1]` requires the existence of at most one edge of this type.
2. A qualifier `[1:1]` requires the existence of exactly one edge of this type.
3. A qualifier `[1:n]` requires the existence of at least one edge of this type.
4. A qualifier `[0:n]` is the default. It imposes no constraints at all.

A cardinality constraint of the form `[min:max]` behind the target class or type of an edge type declaration (following the symbol `->`) defines lower and upper boundaries for the bundle of outgoing edges at a single source node. Such a constraint behind the source class (before the symbol `->`) defines lower and upper boundaries for the bundle of incoming edges at a single target node. The declaration of `owns` edges allows

for instance that any LIBRARY node is related to an arbitrary number of Item nodes, and it requires that any Item node is always related to a single LIBRARY node.

Cardinality constraints are not permanently enforced in order to give a user the freedom to construct the desired graph step by step. Currently, cardinality constraints are checked as late as possible, i.e. not before the first attempt to match or traverse an edge of the regarded type.

3.3.3 Standard Attribute Types and Functions

Before introducing node attributes as an essential part of any graph schema definition, we have to explain what kinds of basic types and functions are available for this purpose. Due to the fact that PROGRES has its main focus on the specification of graph types, it gives only limited support for handling attribute values. There are three built-in *standard types* boolean, integer, and string together with the usual operators (functions) for values of these types. They are offered together with the predicate logic operators, arithmetic operators, and string handling operators. Furthermore, it is possible to define new functions over built-in standard functions, but there are no means for constructing new value types over the three built-in types.

It is a general rule that any important data type should be realized as a graph type, i.e. as a set of related node and edge type declarations. Needed new data types of minor importance may be implemented in a so-called host programming language and imported later on. The import of these data types is explained in the following section 3.3.4. Due to lack of space we are not able to provide the reader with a detailed explanation of all built-in types and functions as well as the available means for the definition of new functions. These details may be found in the language reference manual [Sch98].

3.3.4 External Attribute Types and Functions

The previous section introduced the standard attribute types of PROGRES for numeric, character manipulating, and logical expressions. All other types, as for instance the soon needed type Date, must be defined elsewhere in a so-called *host programming language*. This is either C or a C compatible programming language. Type declarations and procedures written in the selected host programming language have to be compiled and provided to the PROGRES execution machinery in the form of a dynamic library, which is loaded at run-time.

Special *import clauses* are used inside specifications to signal the existence of needed external types and functions. These import clauses are not just lists of type and function identifiers, but convey necessary type checking information about the signatures (return value and parameter types) of imported functions. The following figure 3.4 continues our library example with an import of the above mentioned type Date, a function to generate the current day's date and a function to return the difference between two dates as the number of passed days. Furthermore, the specification fragment imports a unique identifier number generator, which will be used to initialize the id attributes of created DB_OBJECT nodes in the following section.

```
from Time import
  types Date;
  functions currentDate : -> Date,
            days        : ( Date, Date) -> integer;
end;
from ID_Generator import
  functions getUniqueId : -> integer;
end;
```

Figure 3.4: Import of external type definitions.

3.3.5 Intrinsic Node Attributes

Based on previously introduced standard as well as externally defined attribute types we are now prepared to introduce node attributes. They are an essential part of any graph schema declaration and offer the appropriate means to model atomic properties of node objects. Please note that edges may not possess attributes due to the fact that they are just binary relations between nodes without an own identity. As a consequence, attributed relationships (and n-ary relationships) have to be modeled as nodes of an extra type or class, which are connected to their associated nodes via separate edges.

There are three different kinds of attributes. There are so-called *intrinsic* attributes, which have a value of their own that does not depend on values of other attributes. Later on, we will introduce *derived* attributes, which determine their values by means of directed equations from other attributes (cf. section 3.3.6). The third kind of attributes, so-called *meta* attributes of node types, offer means to handle node properties, which have the same value for all instances of a given node type.

```
node class DB_OBJECT
  intrinsic key id : integer := getUniqueId;
end;
node class LIBRARY is a DB_OBJECT
  intrinsic bookLimit : integer := 10;
end;
node type LendingLibrary : LIBRARY end;
node type ReferenceLibrary : LIBRARY
  redef intrinsic BookLimit := 0;
end;
node class TITLE
  intrinsic index title : string;
            authors : string [0:n] := nil;
            editors : string [0:n] := nil;
end;
node type Item : DB_OBJECT
  intrinsic loanDate : Date [0:1];
end;
```

Figure 3.5: Declarations of simple, set-valued, and optional intrinsic node attributes.

Figure 3.5 shows the textual notation for the declaration of six intrinsic attributes (cf. figure 3.2 for the corresponding visual representation of the same declarations). The attribute type definitions of these declarations have the same syntax (and semantics) as type definitions of local variables or formal parameter of attribute functions or graph transformations. The type expression `integer` introduces e.g. an attribute which has a single number as its value. The type definition `string [0:n]`, on the other hand, introduces an attribute that has a possibly empty set of character sequences as its value. And the type definition `Date [0:1]` introduces an attribute which is either undefined or has a single `Date` value.

The first attribute declaration of figure 3.5 belongs to the class `DB_OBJECT`. It requires that any `DB_OBJECT` node possesses an `id` attribute (instance) with a single number as value. The initial value for this attribute is computed by calling the external function `getUniqueId`. All nodes of the following `LIBRARY` subclass declaration do not only have an `id` attribute, but possess the additional attribute `bookLimit`. This attribute is used later on to restrict the number of items that may be lent out to a single person simultaneously. Its declaration defines `10` as the initial value, which is valid for all nodes of type `LendingLibrary`. The `ReferenceLibrary` node type definition, on the other hand, redefines the inherited initial `bookLimit` value and introduces the new initial value `0`. This is just one straightforward example of how subclasses of node classes and node types may override inherited attribute value defining expressions. For further details concerning attribute redefinitions and the resolution of *multiple inheritance conflicts* the reader is referred to [Sch98].

The `bookLimit` attribute is one example of an intrinsic attribute, whose initial value should not be changed during a node's life time, i.e. which has the same value for all nodes of a given type. Changing the keywords `intrinsic` and `redef intrinsic` to `meta` and `redef meta` within the attribute's declaration and redefinition would be the proper solution for making this requirement explicit (instead of stating it implicitely through the absence of any operations which assign a new value to the `bookLimit` attribute).

The example of figure 3.6 also demonstrates that it is possible to build indexes over intrinsic and derived attributes. These indexes allow graph transformation rules to determine all nodes with a certain attribute value efficiently. We have to distingush two cases:

1. *Index* attributes are ordinary attributes for which the PROGRES run-time system maintains an index. The index associates an attribute value with an arbitrary number of nodes in the general case. The attribute `title` in figure 3.5 is one example of this kind; there may be two different `Title` nodes which possess the same `title` text.

2. *Key* attributes are a special case of index attributes. Their indexes associate any possible attribute value with at most one node. The attribute `id` in figure 3.5 is one example of this kind; different `DB_OBJECT` have different `id` attribute values.

3.3.6 Derived Node Attributes and Constraints

The previous section introduced intrinsic attributes, which have a type-dependent initial value. Such an initial value may be changed directly by calling an appropriate graph transformation. Furthermore, we introduced so-called meta attributes, which have constant type-dependent values. Derived attributes, the main subject of this section, have one common property with meta attributes: They are not legal targets of explicit assignment statements. Nevertheless, they usually have node instance specific values, which may change as an indirect consequence of performed graph transformations. Their values are determined by means of directed equations only. A directed equation defines the value of a given derived attribute as a function over a number of attributes of the same node or its direct or even indirect neighbors.

The values of derived attributes are automatically kept in a consistent state. For this purpose an *incrementally working lazy attribute evalution* mechanism is used. It is a variant of the two-phase attribute evaluation algorithm in [Hud87] and an integral part of our DBMS GRAS [KSW95]. It works as follows:

1. A static attribute dependency graph is constructed at compile-time. It records triples of the form *(dependent attribute, defining attribute, path expression)*. The path expression components of these triples determine how the owner node of a dependent attribute is connected to the owner node of a defining attribute. (cf. section 3.4.1).

2. A run-time mechanism inspects the static attribute dependency graph and monitors all relevant graph modifications. It invalidates a dependent attribute, whenever one of its defining attributes or a path to one of its defining attributes is affected.

3. Any read access to an invalid dependent attribute triggers the reevaluation of this attribute and thereby the reevaluation of all its invalid defining attributes, too. The new computed value is stored and reused until it is again invalidated by another graph transformation.

Figure 3.6 shows one example of a derived attribute declaration. It defines the `fullTitle` of a `subtitle` node as the composition of three strings with the concatenation operator `&`. The first string is the `title` of its supertitle, the second one the constant `": "`, and the third one its own `title` attribute. The supertitle of a title is determined by the path expression `<-subtitle-`, which traverses a `subtitle` edge against its direction, starting at the given `TITLE` node `self`. The graph schema of figure 3.3 states that the supertitle of a title is either uniquely defined or does not exist. The evaluation of the subexpression between `[` and `|` returns, therefore, either the `title` attribute value of the supertitle with an attached substring `": "` or fails. In the latter case, the attribute computation continues with the evaluation of the subexpression between `|` and `]` and returns the empty string `''` as a default value.

Figure 3.6 does not only contain one example of a derived attribute declaration, but it presents also two examples of *node integrity constraints*. These constraints are introduced together with derived attributes for the following reasons: They are similar to declarations of derived `boolean` attributes, which raise exceptions whenever their value becomes `false`. Usually such an exception terminates a graph transformation process with a run-time error. But we will see later on in section 3.4.6 that PROGRES offers additional means for the definition of repair actions, which catch these exceptions and transform the erroneous graph until

```
node class TITLE
  intrinsic ...
  derived
    fullTitle : string = [ self.<-subtitle- & ": " | "" ] & self.title;
  constraint
    NoSubCycle = not (self in self.( -subtitle-> + ));
end;
node type Borrower : DB_OBJECT
  intrinsic ...
  constraint
    BelowBookLimit = for all Lib := elem ( self.usedLib ) ::
                      card ( self.hasItem ( Lib ) ) <= Lib.bookLimit
                     end;
end;
```

Figure 3.6: Declarations of derived attributes and integrity constraints.

all violated constraints are fulfilled. This section introduces also a more complex form of constraints which belong to the graph as a whole and not to a single node in the graph.

The first constraint `NoSubCycle` excludes the existence of `subtitle` edge cycles of arbitrary length in the graph. It uses the path expression `-subtitle-> +` to compute the set of all direct and indirect subtitles of a regarded title and requires that the regarded title `self` is not an element of this set. The second constraint guarantees that a `Borrower` observes the book limits of all used libraries. It requires that the number of borrowed items (cardinality of the corresponding set of nodes) from a certain library is always below the value of the `bookLimit` attribute of this library. It uses a seperately defined path expression `usedLib` to determine the set of all relevant libraries and another path expression `hasItem` to compute for each selected library `Lib` all books lent to the regarded `Borrower` node `self` (cf. section 3.4.1).

3.4 Graph Queries and Graph Transformations

The graph schema definition part of a specification as introduced in the last section enables us to specify static properties of a class of DIANE graphs. Using these graphs as the internal representation of a database implies that all database manipulating operations can be described by subgraph selection and (sub-)graph transformation steps. Analogously to the use of the term "schema", in the sense of a database schema, these operations are called "transactions" in the sense of database transactions. Such a transaction is usually composed of basic subgraph matching and transformation steps, which are specified by means of so-called subgraph tests and graph transformation rules.

The following section 3.4.1 starts with the explanation of path expressions (and restrictions), which were already used for the declaration of derived attributes and which are even more important for the definition of complex application conditions for subgraph tests and productions. The following section 3.4.2 introduces our basic means for the definition of parametrized subgraph tests. Section 3.4.3 explains afterwards the basic concepts of productions. Advanced features for defining optional or repeated subgraph patterns as well as for (pseudo-)parallel graph transformations are postponed to the following section 3.4.4. The last section 3.4.5 shows how complex transactions may be programmed by combining an imperative style of programming with the depth-first search and backtracking programming paradigm of Prolog-like languages.

3.4.1 Restrictions and Path Declarations

Restrictions and path declarations — as shown in figure 3.7 — are our basic means for the definition of derived node sets with certain properties and derived relationships between nodes. They are used within attribute evaluation rules to determine context nodes with needed attribute values and within tests and productions as application conditions. The path `usedLib` defines e.g. a derived directed binary relationship between `Borrower` nodes and `Library` nodes. It relates any `Borrower` node to all those `Library` nodes which are reachable by traversing first a `borrows` edge from source to target and then an `owns` edge from target to source.

```
path usedLib : Borrower [0:n] -> LIBRARY [0:n] =
     -borrows->
  &  <-owns-
end;
restriction overdue( DayLimit : integer) : Item =
  valid ( DayLimit < [ days( currentDate, self.loanDate)
                     | 0 ] )
end;
```

Figure 3.7: Textually defined paths and restrictions.

The last declaration of figure 3.7 is an example of a rather trivial parametrized restriction. It restricts a set of `Item` nodes to those nodes, where the number of days between the `currentDate` and the node's `loanDate` exceeds the given limit `DayLimit`. Its definition does not traverse edges (for imposing context restrictions), but inspects a single attribute of a regarded node. Due to the fact that the inspected attribute `loanDate` was defined as an optional attribute (cf. section 3.3.5), it may happen that the subexpression `self.loanDate` and, as a direct consequence, its enclosing expression `days(currentDate, ... )` has an undefined value. This is the reason why we had to use the construct `[ days(...) | 0 ]` to define `0` as a default value, which is is returned instead of an undefined `days` result and which is always below the required day limit.

The `usedLib` path declaration contains a textual *path expression* which navigates from a given set of source nodes along certain edges in a graph and returns their target nodes. The notion of such a path expression within the area of graph grammars was originally introduced to determine and manipulate the embedding of transformed subgraphs [Nag79b]. They are nowadays more important for defining complex application conditions, derived graph properties, and integrity constraints.

Basic path expressions were already used in section 3.3.6 for the definition of attribute evaluation rules. They belong to one of the following categories:

- edge operators of the form `-e->` or `<-e-` allow the traversal of `e` labeled edges from source to target or vice versa,
- attribute conditions of the form `valid exp`, as used within the restriction `overdue`, require that a given node on a path has certain attribute values,
- and restrictions of the form `instance of X`, where `X` is a node class or type identifier, require that a node belongs to a certain type or class.

Many path operators are available for composing more complex expressions from basic ones. Often used operators are the concatenation `p1 & p2` for evaluate first `p1` and apply then `p2` to the result of `p1`, `p1 or p2` for evaluate both `p1` and `p2` and construct the union of their results, `p+` for computing the transitive closure of `p`, and so forth. Furthermore, PROGRES offers appropriate means for the definition of conditional as well iterating path expressions:

- A *conditional* path expression of the form `[ <-owns- | <-subtitle- ]` makes first an attempt to traverse an `owns` edge in reverse direction. If this attempt fails another attempt is made to traverse a `subtitle` edge in reverse direction.
- The *iterating* expression {`not instance of Series :: <-subtitle-`} traverses `subtitle` edges from target to source as long as all nodes on the path are instances of node type `Series`. The path expression returns the end points of the graph traversal, but not their visited predecessors (in contrast to the transitive closure mentioned above).

Furthermore, all visual constructs available for the definition of subgraph tests and left-hand sides of productions may be used for the declaration of paths and restrictions. It is, therefore, possible to use graph patterns for the definition of restrictions and paths, which are used for the definition of more complex graph patterns, which ....

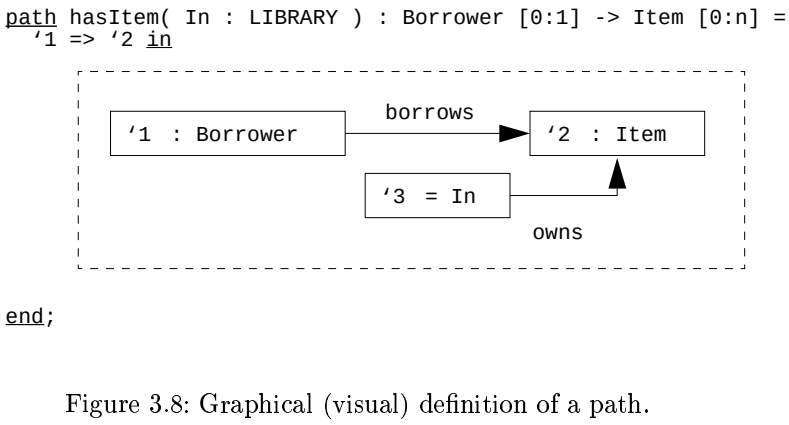


Figure 3.8: Graphical (visual) definition of a path.

The path `hasItem` of figure 3.8 is one example of a parametrized path, which is defined using the probably more readable graphical instead of the more compact textual notation. It connects a `Borrower` node x to all those `Item` nodes of a given library `In` which are borrowed by x. Its graphical definition is evaluated as follows:

1. Determine a start node and bind this node to node (pattern) `'1` of the depicted graph pattern.
2. Traverse all `borrows` edges, which have the match (occurrence) of `'1` as source and bind their target nodes one by one to node (pattern) `'2`.
3. Check whether the node bound to `'2` is the target of an `owns` edge that has the given node `'3 = In` as source.
4. Return to step (2) and continue with the next possible occurrence of `'2` if the check of step (3) fails, continue with step (4) otherwise.
5. Add the occurrence of node `'2` to the result set and return to step (2).
6. Return the computed result set after all possible occurrences of node (pattern) `'2` are processed.

3.4.2 Subgraph Tests and Attribute Conditions

Paths and restrictions are the appropriate means to define derived relationships or functions which take a set of nodes as input and return a set of nodes as output. *Subgraph tests* have to be used whenever one has to define a graph query witout any parameters at all or with more than one (main) input or output parameter of type node set. The test `ReliableBorrower` of figure 3.9 takes for instance a node of type `Borrower` and another node of class `LIBRARY` as input. It has no output parameter except the implicitely defined `boolean` result which signals success or failure of the constructed test. The test checks whether a given person borrowed an item of a given library more than `100` days ago. It succeeds if such an item does *not* exist, it fails otherwise. This behavior was defined using the negative (crossed-out) node (pattern) `'2` which is related to the positive node (pattern) `'1` via the path `hasItem` of figure 3.8.

The node inscription `'1 = aPerson` requires that the source node of the path `hasItem` is the provided `Borrower` node, the path itself restricts the set of possible occurrences of `'2` to all `Item` nodes which belong to the given library `aLib`. The restriction `overdue` of figure 3.7 is finally used to select all those borrowed items with a `loanDate` more than `100` days ago. Please note that a hollow fat arrow between two nodes requires the existence of a certain path (derived relationship) between these two nodes, whereas a hollow fat arrow attached to a single node requires that its target node fulfills a certain restriction (belongs to a derived node set).

`ReliableBorrower` is one example of a test whose single purpose is to check for the existence of a certain graph pattern. It either succeeds or fails. This is indicated by the qualifier `[0:1]` between its parameter list and the equals sign. The following test `FindItem` of figure 3.10 has a different qualifier `[0:n]` due to

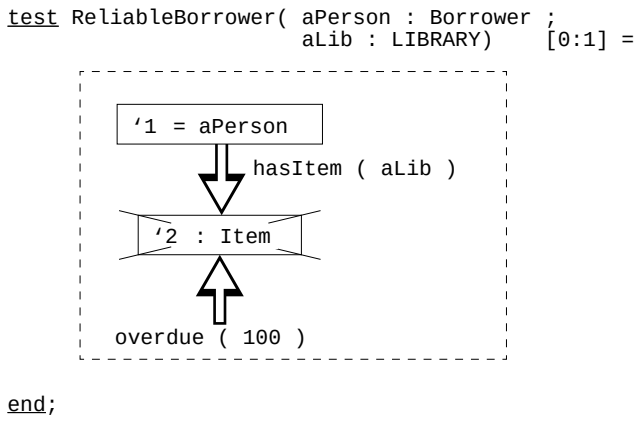


Figure 3.9: A parametrized subgraph test without any output parameters.

the fact that a call of `FindItem` either fails or returns a nondeterministically chosen result. The test takes a (nonempty) set of `LIBRARY` nodes plus a single keyword as input and returns a single node `Match` as output. This node is a nondeterministically selected element of the following set of `Item` nodes: Its elements belong to a library that is an element of the `InLibraries` set and they have a title that contains the given `Keyword` as a substring. The first condition mentioned above is guaranteed by the node inscription '1 = elem(InLibraries), the second one by the attribute condition after the keyword condition.

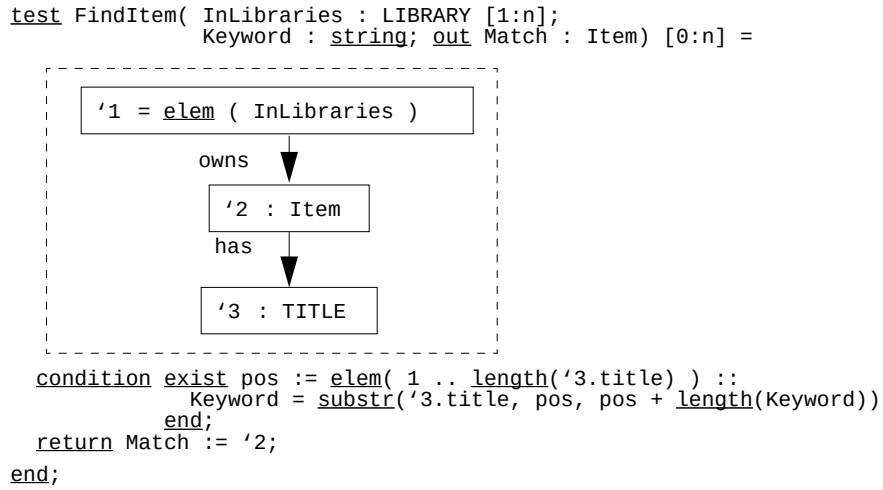


Figure 3.10: A subgraph test with nondeterministic behavior.

3.4.3 Productions and Attribute Assignments

Subgraph tests together with path declarations and restrictions are the basic constructs for inspecting already existing graphs, whereas graph transformations rules — usually called *productions* — are their counterparts for creating and modifying schema consistent graphs. Productions have a left- and a right-hand side graph pattern (LHS and RHS) as their main components, where the LHS may contain the same elements as the subgraph patterns of already presented tests (or paths and restrictions with graphical definitions). A production's RHS has a rather different structure. It may neither contain negative nodes and edges nor restrictions or paths as application conditions.

The production `AddTitle` of figure 3.11 has a rather trivial LHS and RHS, the dashed rectangles above and below the separator `::=`. Its LHS requires the nonexistence of a node which belongs to the abstract class `TITLE` and which has a `title` attribute with the given value `TitleText`. Its RHS creates a new node of type `TitleType`. Permitted actual values for the formal parameter `TitleType` are all those node types which have `TITLE` as a superclass (as e.g. `Book` or `Magazin`). The new node's attributes `title`, `editors`, and `authors` receive their values from the corresponding formal parameters `TitleText`, `EditorSet`, and `AuthorSet`. Any call of `AddTitle` either fails or produces a deterministically defined result. As a consequence, the production has the qualifier `[0:1]`.

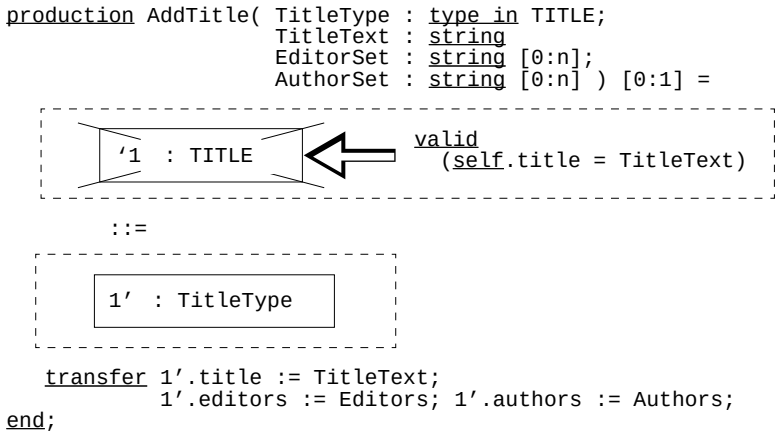


Figure 3.11: A node-creating production (graph transformation).

The production `AddTitle` adds one node to the graph without creating any connections to already existing nodes. The following production `LendItem` shows how it is possible to create edges between already existing nodes or edges between old nodes and new nodes. The production's RHS contains one node with inscription `n' = 'n` for any positive LHS node `'n`. Binding LHS nodes to RHS nodes allows one to distinguish the following three cases:

1. Occurrences of LHS nodes which are not bound to RHS nodes are *deleted* together with all adjacent context edges.
2. Occurrences of LHS nodes which are bound to a uniquely defined RHS node are *preserved* together with all those attribute values and context edges which are not explicitely manipulated by the regarded production.
3. Any RHS node without a corresponding LHS node leads to the *creation* of a new node.

This approach has been adopted from the algebraic graph transformation approach, as presented in [Roz97]. It allows one to copy or redirect a fixed number of edges, but fails if an a priori unknown number of context edges has to be manipulated. PROGRES offers for this purpose so-called *embedding rules*, which were adopted from the algorithmic approach in [Nag79b] and which allow one to manipulate sets of context edges. An embedding rule of the form

```
redirect -borrows-> as <-lost- from '1 to 4' ;
```

deletes e.g. all `borrows` edge which have the occurrence of the LHS node `'1` as source. It creates for any deleted `borrows` edge to a context node n a new `lost` edge which has the context node n as source and the occurrence of the RHS node `4'` as target. These textual embedding rules destroy to a certain extent the visual flavor of graph transformations. Other graph transformation approaches, as e.g. the X graph grammar approach, offer a graphical notation for the definition of embedding rules. The advanced set pattern matching concepts of the following section 3.4.4 gives similar support for the graphical definition of embedding rules.

Based on these general remarks it is easy to explain the behavior of production `LendItem` in figure 3.12. Its LHS matches a subgraph, where the given node parameters `ForPerson` and `ThisItem` are bound to LHS nodes ‘`1` and ‘`2`, respectively. Furthermore, ‘`3` is bound to the always existing and uniquely defined `LIBRARY` node that owns the regarded `Item` node, if this node is an instance of the type `LendingLibrary`. Finally, the LHS forbids the existence of a `borrows` edge between the occurrences of ‘`1` and ‘`2` as well as the existence of another `Borrower` node with a `borrows` edge to the selected `ThisItem` node. The associated attribute condition (below the production's RHS) guarantees that the book limit of the involved library is observed.

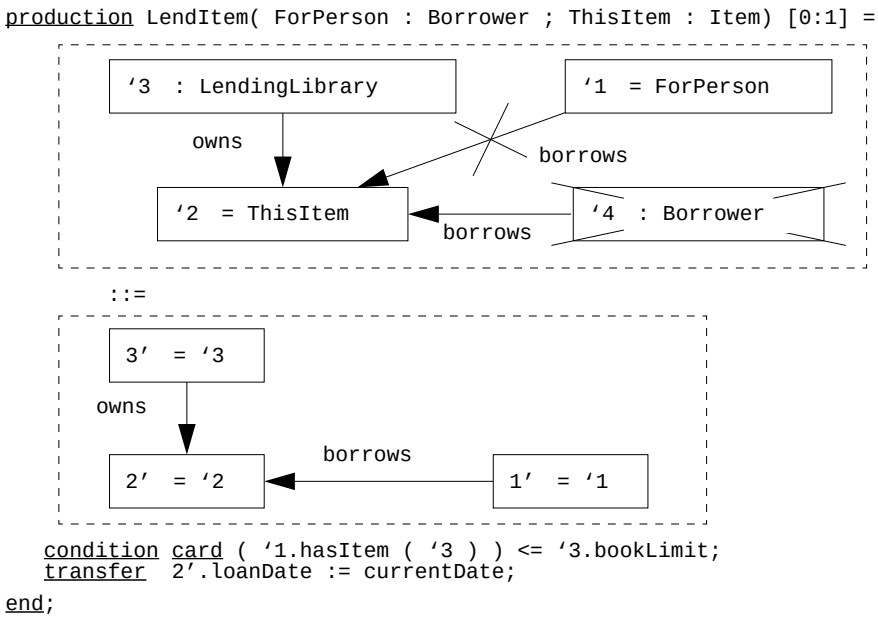


Figure 3.12: An edge-creating production (graph transformation).

A call of production `LendItem` either fails or creates a `borrows` edge between the occurrence of ‘`1` and ‘`2` and assigns the result of the external function call `currentDate` to the attribute `loanDate` of the occurrence of ‘2. The production has no accompanying embedding rules, i.e. it preserves all already existing context edges of the manipulated three node occurrences.

3.4.4 Advanced Pattern Matching Concepts

The production `RemoveOverdueBorrower` of figure 3.13 selects any `Borrower` node which did not return a lent-out `Item` within `100` days. It fails if such a node does not exist, it makes a nondeterministic selection if more than one node fulfills the condition. The production removes the selected node and returns its name as well as an always nonempty set of all related `Item` nodes. This set of `LostItems` is determined by computing the union of the occurrence of the regular RHS node `2’ = ‘2` with the set of occurrences of the dashed double rectangle RHS node `3’ = ‘3`. The dashed double borders of the LHS node ‘`3` and its related RHS node `3’` indicate that it may not only be bound to a single graph node, but to the maximum set of all nodes fulfilling all stated requirements. In the same way as we had four different options for edge type and variable/parameter cardinalities, four different kinds of nodes are available on LHS and RHS of productions:

- Solid simple boxes are *mandatory* LHS/RHS nodes, which have single graph node instances as occurrence,
- dashed simple boxes are *optional* LHS/RHS nodes, which match a single graph node with the required properties if existent, but do not cause failure of the overall matching process if such a node does not exist,

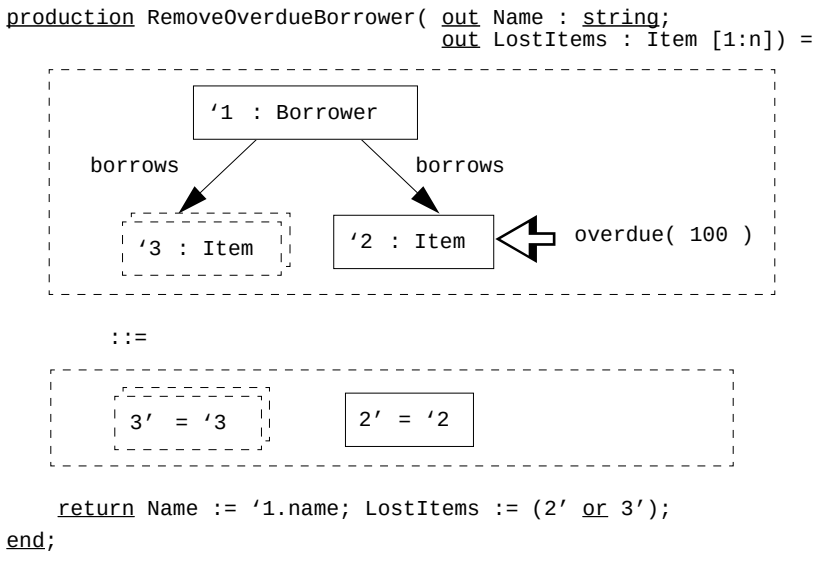


Figure 3.13: A node deleting production (graph transformation).

- dashed double boxes are *optional* LHS/RHS *set* nodes, which match a possibly empty set of graph nodes such that any node of the given set fulfills all mentioned requirements, and
- solid double boxes are *mandatory* LHS/RHS *set* nodes which are mapped onto a nonempty set of graph nodes.

Taking all these different kinds of LHS/RHS nodes into account, we have to extend the overall proceeding for finding matches of a production's LHS as follows:

1. The first step is to find matches for all obligate LHS nodes, such that all positive LHS requirements are taken into account and no two LHS nodes match the same graph node (isomorphic subgraph matching).
2. The next step computes maximal matches for all obligate LHS set nodes, such that additional conditions and the isomorphism requirement are fulfilled.
3. The third step is to find appropriate occurrences for all optional LHS nodes if possible, such that additional conditions and the isomorphism requirement are fulfilled.
4. The fourth step deals with optional LHS set nodes. They are mapped onto a possibly empty but maximal set of graph nodes, such that all additional conditions and the isomorphism requirement are fulfilled, too.
5. Finally, one negative LHS node or edge after the other is handled as follows: the algorithm checks that the already determined LHS occurrence is not extendible with a match of the regarded negative node or edge. Again, we have to disregard those graph nodes which are part of the determined LHS occurrence.

The usage of optional and/or set nodes on a production's RHS is restricted to the identical replacement of nodes, i.e. these nodes are always bound to equivalent nodes of the rule's LHS. It makes no sense to require the creation of a set of nodes of undetermined size in a production's RHS. Additional much more complex mechanisms — such as rule amalgamation in [TB94] — would be necessary to handle situations like "duplicate all nodes which are occurrences of a LHS set node" within one production.

There are two possibilities how to solve graph transformation problems which require the transformation of an a priori unknown number of occurrences of a production's LHS, beside those simple cases which can

be handled by means of set nodes only. The first solution is based on a restricted form of *parallel graph rewriting*. It works if all regarded occurrences of a production's LHS share only preserved nodes and edges. The production `RemoveOverdueBorrower` fulfills this condition and may be applied to all its occurrences in parallel. A single parallel graph transformation step removes then all unreliable `Borrower` nodes, returns the set of their names, and the set of all related `Item` nodes. The declaration of `RemoveOverdueBorrower` has to be changed as follows for this purpose: First of all we have to replace its nondeterministic behavior qualifier `[0:n]` with the parallel application qualifier "`*`". Furthermore we have to change the declaration of its output parameter `Name` such that it returns a set of `names` instead of a single name of the removed `Borrower` node(s).

The second solution for manipulating sets of LHS occurrences is based on iterating control structures. It is the main subject of the following section.

3.4.5 Control Structures and Transactions

It is often necessary to regard rather complex graph queries or transformations which may not be specified as a single subgraph test or production. A pure rule-oriented approach forces one to extend the LHS and RHS of productions in such a way that additional marker nodes guarantee certain sequences of graph transformation steps (applications of productions). This was the main reason for introducing so-called *programmed graph grammars* many years ago, where imperative control structures enforce certain orders of production applications [Bun79]. This approach leads to more readable and better maintainable specifications.

PROGRES goes one step further on in the following sense: It does not only offer imperative deterministic control structures for programming complex graph transformations, but allows one to construct them using a Prolog-like depth-first search and backtracking programming style. It offers for this purpose the following control flow operators:

- The operator `&` corresponds to the left/right evaluation of Prolog clauses. It allows one to define sequences of graph transformation steps.
- Another control structure `choose` ...`else` ...`end` corresponds to the top/down selection of Prolog clauses with matching heads. It allows one to incorporate conditional branches in the program flow.
- Furthermore, the operator `and` is available as the nondeterministic version of the operator `&`. Its arguments are two subprograms which have to be executed in a randomly selected order. Later on backtracking may cause a permutation of the selected order.
- The operator `or` is the nondeterministic counterpart of the `choose` operator. It selects one of its arguments, a subprogram, randomly and executes it. Later on backtracking may enforce the selection and execution of another argument.
- Finally, recursion or conditional iteration as a shorthand for tail recursion can be used to program even more complex graph transformations with the usual risk of nonterminating evaluation processes.

Figure 3.14 contains two examples of transactions. They have the same characteristics as single productions:

- They are *atomic*, i.e. any execution attempt either succeeds as a whole or aborts without any graph modifications.
- They are *consistency preserving*, i.e. they manipulate schema consistent graphs only which fulfill all defined integrity constraints.
- They make their own *nondeterministic choices* and initiate backtracking when a particular decision leads into a dead-end later on.

```
transaction CleanDB [1:1] =
  use Name : string;
      LostItems : Item [1:n] do
    loop
      RemoveBorrower( out Name, out LostItems )
    & write( Name )
    & for all i := elem ( LostItems ) do
        RemoveItem( i )
      end
    end : [1:1] (* loop as a whole is deterministic *)
  end
end;
transaction FindAndLend( InLibraries : LendingLibrary [1:n];
                         Keyword : string;
                         ForPerson : Borrower ) [0:n] =
  use Match : Item do
    choose
        FindItem( InLibraries, Keyword, out Match )
      & LendItem(  ForPerson, Match )
    else
        FindItem( InLibraries, Keyword, out Match )
      & ReserveItem( ForPerson, Match )
    end
  end
end;
```

Figure 3.14: Graph transformations programmed as backtracking transactions.

As a conseqence, transactions of this chapter fulfill the well-known *ACID properties* of database transactions: Atomicity, Consistency, Isolation, and Duration. The underlying database system GRAS [KSW95] is responsible for guaranteeing atomicity, isolation, and duration of transactions, whereas the PROGRES run-time system offers appropriate mechanisms for checking a graph's consistency with respect to all defined integrity constraints (cf. section 3.4.6).

The first transaction of figure 3.14, `CleanDB`, defines a never failing graph transformation with a nondeterministic internal behavior but a deterministic result. Its outer loop is repeated until the execution of its body fails, i.e. until the execution of production `RemoveOverdueBorrower` fails (cf. figure 3.13). This production removes a randomly chosen `Borrower` node from the database, who did not return its loaned items. It fails if such a node is not part of the database and terminates the surrounding loop.

A successful call of `RemoveOverdueBorrower` returns the `Name` of the removed node and the set `LostItems` of all its borrowed items. The transaction prints the `Name` of the removed `Borrower` node using the external function `write` and removes one `LostItem` after the other from the database, too. Please note that the body of the loop has a nondeterministic behavior, but the transaction as a whole has a deterministic behavior. It removes a well-defined set of nodes from the database. It is, therefore, possible to qualify the whole transaction as well as its main loop with `[1:1]`. The qualifier `[1:1]` overrides the default qualifier `[0:n]` for nondeterministic partially defined transactions. It provides the reader of a specification as well as the PROGRES compiler and interpreter with some extra information about a constructed transaction's behavior. This information is used for checking a specification's consistency and for avoiding unnecessary bookkeeping overhead for backtracking purpose.

The second transaction of figure 3.14, `FindAndLend`, is one example of a graph transformation, whose implementation relies on *depth-first search and backtracking*. Its body is a `choose` construct with two branches. Both branches contain calls to nondeterministic as well as partially defined productions. The transaction's execution starts with the call of `FindItem` in the `choose` statement's first branch. This call either fails and causes the execution of the `choose` statement's second branch or returns an `Item` node which belongs to one library in the set `InLibraries` and which has the given `Keyword` as a substring in its title. The `LendItem` production call takes the selected `Match` as input. It fails if the selected `Match` is already lent out to another person and triggers backtracking. Backtracking returns to the last nondeterministic selection in the flow of control and restores the manipulated graph's old state if necessary. In our case we have to reexecute the test `FindItem` such that it returns the next matching `Item` node of the graph. The execution continues with another call of `LendItem` if such a node does exist, it aborts the execution of the first `choose` statement branch otherwise.

The second branch of transaction `FindAndLend` reinspects all matching `Item` nodes and makes a reservation for the first match. A more realistic implementation of `FindAndLend` would inspect the set of all matches and select an element with the earliest `loanDate` and/or the lowest number of reservations.

3.4.6 Consistency Checking with Constraints

Graph tests may be used in a straight-forward way for supervising the structure and state of a manipulated graph. They can be embedded in the control flow of transactions in order to check for certain graph patterns which are considered essential for the consistency of the underlying graph. Depending on existence or non-existence of these patterns it is possible to proceed with different cases in the specification. The disadvantage of this approach is that the functional behavior of the specified system gets lost in graph tests and branching to control flow alternatives.

It depends on the purpose of a given specification whether it is necessary to check the consistency of the graph structure before or after a certain graph transformation explicitely. These checks allow one to recognize unexpected or unwanted situations and to initiate backtracking as one possibility to return to a consistent graph state. Nevertheless, the initiation of backtracking alone does not necessarily indicate an error situation. It is rather the strength of PROGRES to let its users decide whether a problem is better modeled adhering to a declarative or imperative programming style.

Furthermore, it is a matter of early specification phases to check the well-definedness of graph state changes. As long as the complete specification is under construction a debugging facility is useful which helps to identify and to eliminate errors. At a later time, when the specification has proven to work sufficiently correct, it should be possible to disable the checking mechanisms for reasons of efficiency. In that case they can be considered as assertions or extended documentation. Furthermore, they may be used for verifying the correctness of critical operations.

Integrity constraints have been introduced in database systems as a mechanism to ensure consistency of their large data sets. In [HW95] the integration of graphical integrity constraints with graph transformation based on category theory was proposed. We have adopted the proposal for the purpose of checking graph consistency during the execution of PROGRES specifications.

Node integrity constraints were already introduced in section 3.3.6 because of their close relationships to node attributes. Node integrity constraints are used to monitor the state of single nodes (and their direct neighborhood). They are not very helpful for examining the graph as a whole and for stating more complex consistency conditions. Therefore, PROGRES offers *global integrity constraints* which can either be defined using a textual or a graphical notation.

In its simplest form, such a constraint consists of two graphical parts, the `for` and the `ensure` part. The semantics of the constraint is that each pattern which matches the `for` part in the graph must also be extendible to match the `ensure` part. Otherwise the constraint fails and the execution is halted. The `for` part of the constraint is optional. In that case the occurrence of the `ensure` pattern in the graph is checked. That means the semantics of the constraint changes from "for all" to "exists" quantification.

Integrity constraints are very helpful for monitoring and preserving the internal consistency of a given specification. They may for instance be used to require that whenever an `Item` has a `Borrower`, the `Item` has to belong to a `LendingLibrary` and, furthermore, that there is at most one `Borrower` at a given time.

Figure 3.15 defines an appropriate integrity constraint. It forbids the existence of more than one `Borrower` node associated with the same `Item` node. Furthermore, the constraint ensures that only `LendingLibrary` items are lent out, whereas `ReferenceLibrary` items must remain present. A consistent `LIS` specification may never violate such a constraint. Any attempt to violate the introduced constraint terminates the execution of the erroneous graph transformation instantly. In early stages of development this kind of constraints is a useful debugging facility.

In some cases it would be helpful if a specification would not only be able to check whether all defined constraints are respected (and to stop the execution otherwise), but also to remove detected inconsistencies on its own.

Active constraints have been invented for this purpose [WC96]. They extend the notion of integrity conditions by so-called *repair actions* which are executed if a certain constraint is violated. That means, where

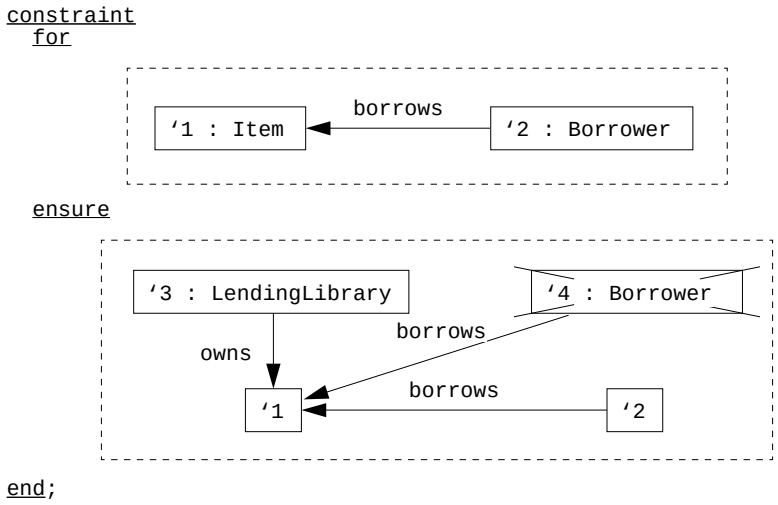


Figure 3.15: Integrity constraint in the Library Information System.

the internal consistency checking constraint halts execution, the active constraint "repairs" the forbidden graph state. Note that if the constraint is still violated after the repair action has been executed the execution halts for this kind of constraints, too.

Figure 3.16 shows a constraint with a repair action which is rather similar to the operation `RemoveOverdueBorrower` in figure 3.13. For every pair of borrower and borrowed item that can be matched, it is checked whether the lent item is overdue. If overdue items are found the operation `RemoveThisBorrower` is called, which removes the borrower from the set of library users. The books borrowed by the removed user are collected in a "lost items" list.

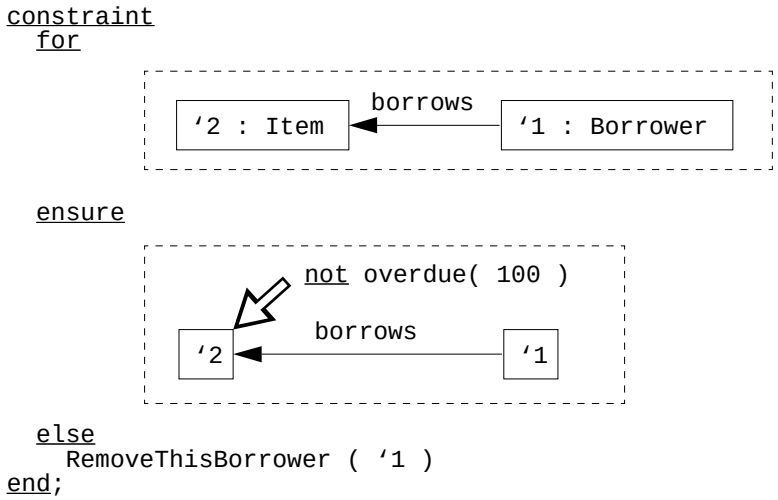


Figure 3.16: Constraint with repair action.

As a consequence constraints can be used for two different purposes:

- "Real" integrity constraints are used to detect invalid states of the underlying model with respect to required *invariants*. They represent the run-time counterpart of the static type system of PROGRES. That means, situations which cannot be handled statically need to be noticed and suppressed during run-time. These constraints do not repair, they merely detect. They are used for debugging purposes

during development of the specification and can be turned off for tested correct specifications without changing the behavior of the system.

- Constraints with repair action are used to automatically call operations that react to an explicitely modeled *trigger*. The trigger describes a forbidden graph state that demands reaction as soon as it is detected. It is a matter of the model whether it must be handled immediately, or not. Repair actions can be regarded as comfortable means to let the specification handle tasks automatically which otherwise would have to be included as "test and repair" in the control flow at many places in the specification. These constraints can be seen as coupling mechanism between states and operations. Of course, these constraints cannot be turned off without affecting the functionality of the system.

Sometimes it is a matter of taste whether a regarded consistency condition is treated as a real integrity constraint or as an active constraint with a repair action. One might for instance argue that an `Item` instance associated with two `Borrower` nodes constitutes a severe integrity constraint violation, which should even be respected by any intermediate and temporarily existing graph state of a complex transformation process. This point of view was adopted in figure 3.3, where the edge type `borrower` was declared together with the cardinality constraint `[0:1]` for its `Borrower` target nodes. In this case the `LIS` specification has to be debugged whenever the constraint is violated, until we have identified the graph transformation which associated a second `Borrower` node with one `Item` node.

On the other hand one might argue that the `LIS` graph specification should be able to handle such an inconsistent situation on its own. In this case, an active constraint might be useful, which calls a `Borrower` node removing repair action as a kind of exception handling routine. Active constraints in this sense are closely related to the concept of *ECA rules* in active databases [WC96]. They are Event-Condition-Action triples, where an action is triggered if the condition holds after the occurrence of a specified event. In our terms events are implicitely given by calling a certain operation. Conditions might either be graphical patterns in global constraints or boolean evaluation functions in schema constraints. If they are violated optional repair actions might be invoked.

To summarize PROGRES provides means to ensure the correctness of specifications on different levels.

- Using DIANE graphs as the data model avoids problems with pointers, which are a typical source of errors in imperative programming languages.
- The context-free syntax of specifications is enforced by the editors of the PROGRES environment (cf. section 3.6).
- Hundreds of static semantics rules guarantee that graph elements are only combined in a correct way with respect to the language's type system. To check these rules at compile time is the task of the PROGRES analyzer (cf. section 3.6).
- Edge type cardinalities offer a smooth transition from compile-time type checks to run-time checks. Some productions which violate defined cardinality constraints can be detected at compile-time, but most of them have to be caught at run-time.
- Pre- and postconditions (not presented here) are assertions for graph transformations, which check the well-formedness of a graph before a transformation takes place in order to guarantee its well-formedness after its end. The violation of pre- and postconditions terminates an execution immediately.
- Integrity constraints are able to enforce the overall consistency of a manipulated graph. They have to be checked at run-time, too. Some of them halt the execution if violated, others trigger an appropriate repair action.

At the end of this section we have to admit that we did not discuss the problem, when defined integrity constraints have to be checked. It is obviously impractical to check all constraints after every graph transformation step. On the contrary it must be possible to associate a given constraint with a set of graph transformations. These transformations may produce inconsistent intermediate graph states, but they may not return an inconsistent graph as their final result. This topic is closely related to the main topic of the

following section, the PROGRES module concept. These modules (packages) offer the appropriate means to associate a set of integrity constraints with a set of exported graph transformations that have to respect their constraints.

3.5 Modules and Updatable Graph Views

So far, the rule-based, object-oriented, and imperative programming constructs of PROGRES have been introduced. The specification which we have used as a running example is still rather small. Serious specifications have a size of more than 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. Thus the lack of any implemented module concepts for graph transformation systems is one of the main hindrances for a wider distribution of graph transformation specification or programming languages in general and the language PROGRES in particular.

This problem 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" [Par72] and "Programming–in–the–Large" [DK76] have been invented to overcome this problem. Later on, these concepts have led to the development of modular programming languages like Modula-2 and Ada [WS84], the development of software design languages like HOOD [Rob92], and to the development of module concepts for knowledge base representation languages like PROTOS-L [Bei95].

It is a common goal of the graph grammar community to introduce modular programming and refinement concepts. Therefore, a variety of publications have been presented concerning this issue. They concentrate on different issues of abstraction and structuring of specifications:

- The journal paper [KKS97] introduces functional abstraction with graph transformation units.
- A view concept, which allows merging specifications with a common specification basis, is discussed in [EHTE97].
- A first survey of possible graph transformation module concepts is given in [EE96]. It deals with precise semantic definitions based on module concepts for algebraic specification languages.

To a large degree the problems found in the area of object-oriented modeling are quite similar to the ones experienced with graph transformation systems. We have taken the joint efforts of the best-known object-oriented modeling forces, the "Unified Modeling Language (UML)" [FS97], as a source of inspiration. UML which has become standard provides the concept of *packages*. Packages, i. e. their contents, can be used, nested, and refined. They seem to be a suitable modularization approach which can also be transferred to graph transformation systems. Because of the lacking formal foundation of the semantics of packages and their different relationships, we have precisely reformulated the definitions and restrictions in about a dozen predicate logic formulas [SW97].

Taking together the experiences with modular programming languages, which have been the basis for our first proposal for a PROGRES module concept [WS97], the influences from the graph grammar community, and the package concept proposal of the industrial-strength OO modeling languages, we present a module concept which fulfils the following requirements:

- A package constitutes a static, logical unit of data together with operations of a certain complexity. It has no effects at runtime.
- A dependency between packages reveals some of the server's resources while others remain inaccessible to the client.
- A package can be developed, compiled, and tested independently from other packages and is replacable by a package with the same export interface.

Adopting the basic properties of UML packages was possible due to the fact that the concept of establishing a name space for a collection of declarations and visibility rules for clients accessing other packages are

independent of their contents. Furthermore, with UML being a graphical language, its packages do not separate interface and body. This is an important advantage for PROGRES because even if considered as implementation details, the body of a production is a comprehensive documentation of its behavior. Nevertheless some resources are *public* while others remain *private*. Public elements are accessible in client packages by an *import dependency* from the client to the server package. Private resources are not accessible from outside, at all.

Packages are intended to be used for quite different purposes, varying from the realization of encapsulated abstract data types to the definition of open graph database (sub-)schemata. Together with pre-, post-, and global integrity conditions a package builds a unit with a set of exported resources and semantic properties of the specified graph type. Following the concept of "Design-By-Contract" [Mey97] a package can be reused in different contexts.

3.5.1 PROGRES Packages

The simplicity of the running example and the fact that all working areas are concerned with "doing something with books" makes it possible to cover the structure of the whole example in one single schema. In order to demonstrate how packages can be used in PROGRES specifications the `LIS` example is extended. So far the schema in figure 3.2 comprises different working areas which are rather independent of each other.

1. The *library administrator* is responsible for purchasing new books and entering them into the library catalogue.
2. The *book database manager* keeps the information about books and magazines up-to-date in the central database.
3. The *user adminstrator* issues and revokes library user cards, and handles the lending and returning of books or magazines.

We will now look further into the area of library adminstration. Let us assume that the library administrator has made bad experience with the return reliability of the library's users. Too often books are returned too late. The operation `RemoveOverdueBorrower` in figure 3.13 filters out the borrowers which have lent out books longer than 100 days and kicks them out of the library. Unfortunately, together with the borrower the books are also considered lost. In order not to rebuy non-returned books over and over, it seems promising to kindly remind the borrower to return a book after the permissible borrow time of 30 days. For that purpose, the library administrator needs a list of book items which are in fact borrowed for more than 30 and less than 100 days.

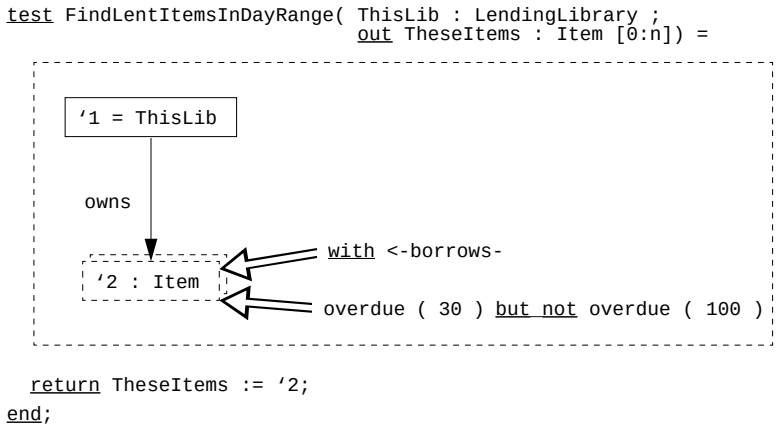


Figure 3.17: Test determining all books lent for a given range of days.

Test `FindLentItemsInDayRange` in figure 3.17 is rather similar to the LHS of figure 3.13. It returns all books of the currently regarded library and filters out all borrowed books by requiring the existence of an

incoming `borrows` edge. The resulting set of books is further reduced by applying the restriction `overdue` in figure 3.7 with different values.

Since the loan date is not maintained by an index there is no efficient access to book items with a given loan date. Even if an index is maintained, the underlying run-time system does usually not support range queries efficiently. That means, to yield all elements matching the values in a range, we have to iterate through the set of all possible candidates and to check whether their values lie in the given range. In order to support range queries directly and to find all items lent out for more than 30 days for the "reminder" list effectively, it is necessary to implement the loan date management by a tailored data structure on the underlying graph.

The management of loan dates can be performed by a binary search tree implementation. The arrangement of data in binary search trees allows the efficient access within a given range of values. A complete specification of binary search trees with PROGRES is rather complex, if the tree is required to be balanced throughout its life time. In this description we do not care about implementation details for the sake of brevity and simplicity [2].

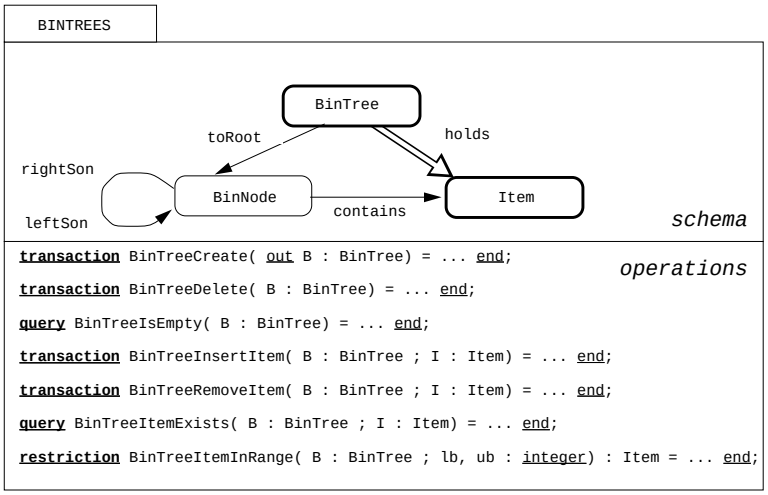


Figure 3.18: The interface of the binary search tree package.

Applying the principle of data abstraction in the classical sense to the binary search tree package `BINTREES` results in figure 3.18. Recall that we actually do not distinguish interface and body of packages. Instead, we distinguish public and private resources. In figure 3.18 the bold elements represent public elements, the others represent private elements. Public resources, correspond to a package's interface, whereas private resources correspond to its body. Public resources are visible via an import dependency, private ones are not. Nevertheless we will also use the terms interface and body in the following.

The internal binary tree implementation in figure 3.18 need not be visible for the library adminstrator operations. Therefore, only the "anchor type" `BinTree` of the package and the entry type `Item` is exported at the interface of the schema part. The type of the internal tree nodes remains hidden together with the tree edges. The operations part provides the typical operations of an abstract binary tree type for creating and deleting trees, for inserting and removing values from a tree, for retrieving values from the tree or testing its emptiness, and, finally, for restricting items to those within a given range of values. These operations are equipped with formal parameters corresponding to the current tree and item for which the operation is invoked. Technically, the types of the formal parameters — `BinTree` and `Item` — must be visible for every operation.

We will now answer the question how to handle different parts of a specification and their mutual dependencies from an architectural point-of-view.

[2] In [Sch94] we have presented a complete specification of balanced binary search trees, so-called red/black trees.

3.5.2 Specification-in-the-Large with Packages

Having the binary tree package with an efficient implementation, on the one hand, and the range query in `FindLentItemsInDayRange`, on the other hand, we have to set the packages `LIBRARY_ADMINISTRATION` and `BINTREES` in relation.

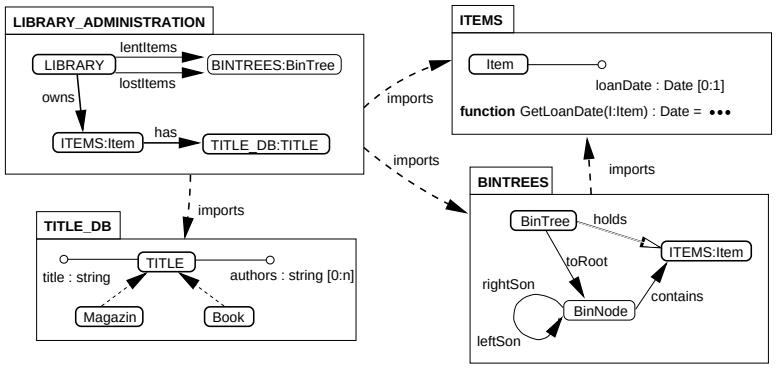


Figure 3.19: PROGRES packages and their dependencies.

Figure 3.19 roughly sketches the dependencies in the `LIS` specification. Obviously, the package `LIBRARY_ADMINISTRATION`, which comprises all the functionality sketched above, uses the binary tree package. This *use dependency* from `LIBRARY_ADMINISTRATION` to `BinTree` is established with an *import*. The concrete dependency on node instance level is realized by introducing two new edge types `lentItems` and `lostItems` from nodes of the class `LIBRARY` to nodes of the class `BinTree`. Both, the set of lent books and the set of borrowed books are maintained using the binary tree implementation. In order to avoid cyclic import dependencies, the node type `Item` is extracted from `LIBRARY_ADMINISTRATION` and is exported by an own package `ITEMS` for both clients. Furthermore, the package `TITLE_DB` is a central component responsible for maintaining the database of all sort of publications. It is used by the package `LIBRARY_ADMINISTRATION`, too. Apart from not being complete, the contents of the packages is identical to the graphical schema in figure 3.2. Additionally, we have used the possibility to hide private resources from external use. Only public resources represented by bold graph elements are visible via the import dependency.

The small "architecture" in figure 3.19 does only illustrate the static dependencies on schema level. The concrete access from the `LIBRARY_ADMINISTRATION` to the `BINTREES` package is established by replacing the `overdue` restriction by a new restriction which makes use of the range query facility imported from the `BINTREES` interface. In this restriction the parameters are directly passed — after evaluating the dates corresponding to the range of days — to the binary tree implementation.

In this special case it seems very natural to use specifications with total data abstraction. This is true because in the test `FindLentItemsInDayRange` in figure 3.17 we have pretended a visual specification with graph transformation rules but, in fact, we have cheated by making the transition to textual procedure calls by using the restriction `overdue`. This might appear adequate for queries, where complex evaluation functions can be hidden in paths, restrictions, and derived attributes within graphical patterns. In general it is the purpose of rules to manipulate the graph structure even of imported packages directly by circumventing their procedural interfaces. The notion of visual programming with graph transformation rules is based on the description of transformations on the level of nodes and edges directly [SWZ95a]. Of course, information hiding and abstraction from internal realizations has proven to be useful in order to ensure that the overall graph structure remains in a consistent state during the execution of specifications.

Going back to rule `LendItem` in figure 3.12 the dilemma of visual programming with graph transformation and data abstraction becomes more obvious. `LendItem` is an intuitive description of the changes in the graph structure during its application. In that way rules are an ideal means to show the effect of an operation using almost the same notation and the same elements as in the graph schema. Therefore, schema and rules fit perfectly together. But, how should a client ensure consistency of the manipulated graph structure with respect to the server's requirements without completely reimplementing its behavior, anyway?

If a book is lent out, it must be inserted as an element of the `lentItems` set maintained by the binary search tree. The `LIBRARY_ADMINISTRATION` client is only interested in creating the `borrows` edge (cf. `LendItem` in figure 3.12). There is no connection between `LendItem` and `BinTreeInsertItem`. How will the insertion be propagated to the binary search tree without knowing the internal realization with e.g. `BinNode`? If we can not give a satisfying answer to this question, we can either forget about data abstraction for graph transformation systems, or we have to dictate interface operations for each access to the graph structure.

This problem has nothing to do with internal consistency of abstract graph types, it has a closer relation to the *view update problem* well-known from databases. That means, abstract graph type packages can be regarded as *views* on the internal graph structure. The overlapping parts of different views correspond to imported schema elements. Consequently these parts are difficult to handle concerning creations or deletions of graph elements. Updates performed "through" a certain view on the graph, which abstracts from the detailed graph structure, will potentially risk to destroy the global consistency of the common graph structure.

There are two obvious possibilities to make the operation `LendItem` use the binary tree interface.

1. The graphical transformation rule is called within a transaction together with the call of the insertion operation in the binary tree specification.

2. The binary tree specification gives up the data abstraction principle and reveals its internal structure to the outside world. In that case the rule would create internal binary tree nodes and edges on its own preserving the search tree property.

Both solutions are not satisfying. In the first case, we have to sacrifice the intuitive character of graph transformation rules. In the second case, rules are blown up with specific implementation details which establish a very tight dependency to the concrete realization because they have to preserve the internal graph consistency. This makes it impossible to exchange the underlying realization by another one, which is one major aspect of the motivation for any module concept. The compromise solution would need to provide means for, on the one hand, preserving the visual specification style with rules, and, on the other hand, allowing data abstraction from implementation details by using interfaces which hide the realization of complex data structures internally. That means, a solution seems promising, where graph transformations are described visually in one package and are mapped on an efficient implementation in another package automatically.

3.5.3 Graphical Modeling with Updatable Graph Views

In section 3.4.6 we have learnt how constraints can be used to keep the underlying graph structure in a consistent state. For packages the idea of consistency and invariants gets a new significance. Considering a package as an abstract graph type with total information hiding, the semantics of the specified graph type should be transparent for the clients. An abstract graph type is more than just a list of available operations. Its semantical properties are essential to capture the true nature of the type's instances. In studying constraints for detecting inconsistencies we have not yet related the notion of integrity to the concept of data abstraction. We consider constraints to be an appropriate means to enforce a certain behavior of packages. In that way, packages "promise" to leave the underlying graph in a consistent state after having invoked its interface resources on a hitherto consistent graph structure. This concept is known as *Design by Contract* [Mey97]. We will now extend this concept in order to resolve the "graph transformation dilemma".

For "Specification-in-the-Small" purposes we have identified two kinds of constraints. We have classified them as integrity constraints (invariants) signaling inconsistency and as active constraints (triggers) invoking repair action automatically. This characterization can be applied for the purpose of data abstraction with packages as well. We have seen that the interface of another package can be invoked either by calling interface operations or by accessing schema resources. It is the decision of the server whether it allows its clients to access its exported graph elements directly or forces them to call exported operations only.

A package insisting on total data abstraction can be sure that its internal view on the common graph remains consistent as far as it is responsible. But, recall that invoking procedure calls contradicts the visual character of graph transformation. Packages with a revealed schema have to ensure actively that their underlying structure is not destroyed if the graph is manipulated directly.

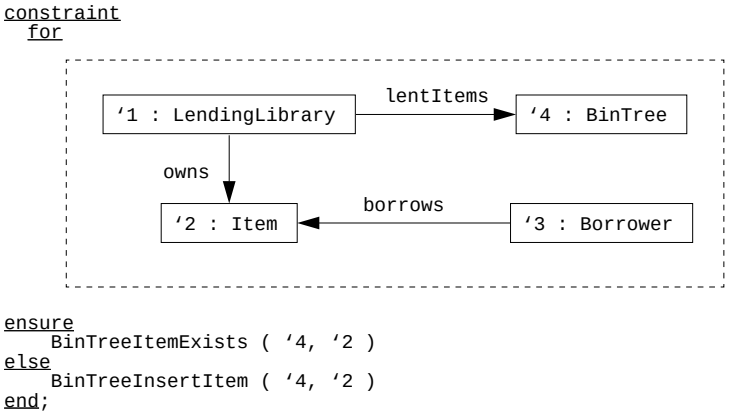


Figure 3.20: Restriction Using the Interface of the Binary Search Tree Specification.

In figure 3.20 we present the coupling between the `Library_Administration` operations and its internal realization with binary trees. Note that the `BINTREE` package is constructed as an abstract graph type following the principle of data abstraction. The active constraint in figure 3.20 checks for every book item that has been lent out. The `for` part is similar to the RHS of the production `LendItem`. Additionally, the relationship to the binary tree package is present in the constraint. According to the semantics of active constraints we demand that for every pattern matching the `for` part the `ensure` part must not fail. Otherwise the repair action in the `else` part is invoked. Note that in this example the `ensure` part contains the call of the interface operation `BinTreeItemExists`. This query checks whether the borrowed item matched in the `for` part is already present in the set of lent out items maintained by `BINTREES`. If this is not the case, the book item is inserted into the binary tree by invoking the repair action `BinTreeInsertItem`. Assuming a correct implementation of `BinTreeInsertItem`, the `ensure` part will not fail again after inserting the item.

Besides the constraint that controls lending out items, there is also the symmetric counterpart for returning items or declaring items lost, resp. which is not shown here.

3.6 The Programming and Prototyping Environment

In the last sections the PROGRES language has been introduced. Having graphs as data model, the expressiveness as a visual programming language has been emphasized. We will now sketch the tool support of the PROGRES environment for modifying, analyzing, and executing graph transformations.

The next subsection gives first a survey of the most important tools and their underlying main data structures. It shows that all these tools are connected with each other such that they form an *integrated* environment. The following three subsections present available tools from a user's perspective.

Finally, we have to emphasize that PROGRES tools form another instance of an integrated (meta) programming environment and are implemented using *IPSEN graph technology*. They use the same basic components and have the same design as all other tools of the software engineering environment IPSEN. For further details concerning the architecture and the realization process of the PROGRES environment the reader is, therefore, referred to [Nag96].

3.6.1 Basic Components and their Interdependencies

Equipped with some knowledge about the language PROGRES, the reader probably can imagine the difficulties we had to face during the development of its programming environment. It consists of the following tools:

- a mixed textual/graphical syntax-directed editor together with an incrementally working unparser and a layout editor for graph schemata, productions, etc.,

- an integrated (micro-)emacs like editor for entering textually represented language constructs together with an incrementally working LALR-parser,
- an incrementally working analyzer which detects and explains all inconsistencies with respect to the language's static semantics,
- an import/export interface to plain text editors and text-processing systems (currently for FrameMaker from Adobe only),
- a hybrid interpreter/compiler which translates first a specification into byte code of an abstract graph transformation machine and interprets the code afterwards,
- an instantiation of the generic ffgraph browser library developed at the University of Passau with simple view definition facilities for monitoring graphs during an interpreter session,
- two compiler backends which translate the graph machine byte code into plain Modula-2 or C code,
- another backend which produces code fragments for the user interface toolkit tcl/tk [Ous94], and finally
- tools for version management and three-way merging of different versions of one specification document.

The most important basic component of the PROGRES environment is the database system GRAS [KSW95]. It supports efficient manipulation of persistent graph structures, incremental evaluation of derived graph properties, nested transactions, undo&redo of arbitrarily long sequences of already committed transactions, recovery from system crashes, and has various options of how to control access of multiple clients to their data structures. In this way, persistency of tool activities including undo&redo and recovery comes for free. Furthermore, integration of tools is facilitated by storing all data within a GRAS database as a set of related graphs.

The *editor* and *analyzer* are those tools of our environment that assist its users when creating and modifying specifications. Both tools are tightly coupled and they are even integrated with the interpreter tool. In this way, the tedious edit/compile/link/debug cycle is avoided and the environment's user is allowed to switch back and forth between editing, analyzing, and debugging activities. The editor itself is not a monolithic tool, but consists of a number of integrated subtools. These subtools support syntax-directed editing as well as text-oriented editing of specifications, pretty-printing, manual rearrangement of text and graphic elements as well as browsing and searching activities.

Figure 3.21 displays the major data structures of the environment and the transformation processes between them. Its right-hand side deals with compiling and executing specifications and is, therefore, specific for the PROGRES environment. Its left-hand side, on the other hand, deals with editing and analyzing activities which are part of any integrated set of IPSEN tools. It shows that specifications are stored in the form of two closely related documents which are internally realized as graphs. The *logical document* contains a specification's abstract syntax tree and all inferred type checking results. The accompanying *representation document* captures all concrete syntax information, including the chosen layout of (nested) text and graphic fragments.

An incrementally working *unparser* propagates updates of the logical document into its representation document. The unparser's counterpart is a *parser* which propagates changes in the reverse direction. It takes a line/column oriented text buffer and not a hierarchically structured representation document as input. Currently, "free" editing of graphical specification fragments with a micro-emacs like text editor is only supported via such a textual representation. In [Nag96] the reader will find a more in-depth explanation of the left-hand side of figure 3.21 and the realization of its components. The right-hand side of figure 3.21 reveals that there are two alternatives how to execute a given specification. The first one is based on direct interpretation. It is mainly used for debugging purposes, when intertwining of editing, analysis, and execution activities is advantageous. The second one is to translate a specification into equivalent Modula-2 or C code for rapid prototyping purposes.

The diagram in figure 3.21 shows that the editor and analyzer play the role of a conventional compiler's front-end and provide all information about a specification's underlying abstract syntax tree and its static semantics. An incrementally working compiler takes this information as input and translates executable

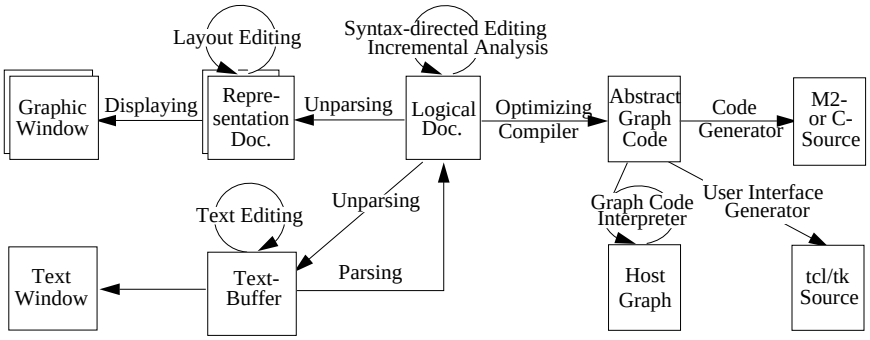


Figure 3.21: Data structures and transformation processes in the PROGRES environment.

increments (on demand) into byte code for an abstract graph transformation machine. The compiler is the most important component of the whole execution machine and determines its efficiency to a great extent. This is especially true in the case of productions, where we have to choose between a large number of different byte code sequences. Any of these code sequences perform the required subgraph matching, but they may differ with respect to runtime efficiency considerably (for further details cf. [Zün96]).

The produced abstract graph code may be executed directly or serves as input for two compiler backends, which produce equivalent Modula-2 or C code. The underlying abstract graph transformation machine combines the functionality of a conventional stack machine with backtracking capabilities. Furthermore, its main component offers facilities for manipulating persistent graphs.

The translation of abstract graph code into readable Modula-2 or C source code is rather straightforward with the exception of backtracking, which requires reversing a program's flow of control, restoring old variable values, and undoing graph modifications. Using undo&redo services of GRAS, the main problem is to reverse a conventional program's flow of control without having access to the internal details of its compiler and runtime system. A description of the problem's solution is beyond the scope of this book, but may be found in [Zün96].

Following the *Graph Grammar Engineering* approach in [SWZ95b], we will now present the tools of the PROGRES environment and their support for various phases of specification development.

3.6.2 Editing and Analyzing Specifications

A specification for a certain system is initiated by considering different scenarios. They help to identify node objects, their relationships, and the operations which perform structural changes on the graph during run-time. The static structure of well-formed graphs is established in the graph schema. The graph schema for the `LIS` example has been introduced in section 3.3.2.

The PROGRES environment provides two views on the graph schema, the textual and the graphical view. The two views in figure 3.22 present the schema as shown in figure 3.2 and 3.3. The graphical schema editor allows to define the node class hierarchy, to assign attributes to node classes, and to add relationship edges between them. The presentation is known from ER diagrams. The textual view is used for attaching evaluation functions to node attributes and constraints, which is outside the scope of the graphical view.

Specifying visually is not only restricted to the graphical view of the schema. Graphical elements are also the basis of the operational part of PROGRES. In contrast to textual languages, *structural* or *syntax-directed editing* is a useful means for writing graphical patterns in graph transformation rules. Nevertheless, since PROGRES is a hybrid textual/visual language it is also necessary to provide means for writing several lines of text with a comfortable text editor. The common basis for both, textual and graphical representation is the abstract syntax tree. The syntax-directed editor allows to expand the nonterminals of the language step by step until concrete terminal symbols of the language are reached. The textual fragments are parsed and presented as if they would have been expanded with the syntax-directed editor itself. All graphical language elements do also have a textual counterpart which is e.g. used for writing specifications in a file.

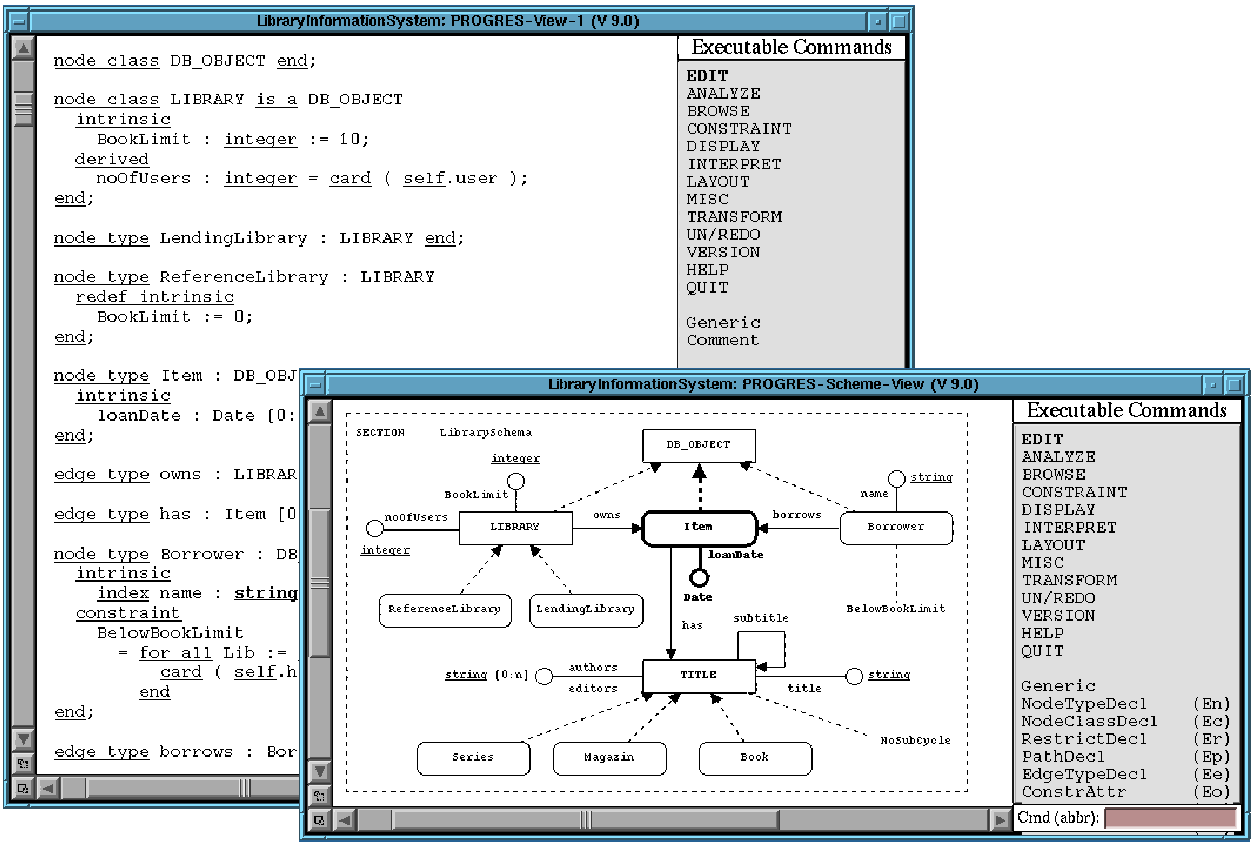


Figure 3.22: Graphical and textual view on the graph schema.

PROGRES is a statically typed language. To avoid type errors as soon as possible the *incremental analysis* tool is integrated into the editor. Analysis of PROGRES specification comprises hundreds of static semantics rule. They check the consistency of a specification among other things with respect to

- *identifier binding rules*: every applied occurrence of an identifier must be bound to a visible proper declaration, any identifier declaration should have at least one applied occurrence
- *inheritance rules*: multiple inheritance requires the hierarchy to form a lattice and to search for unresovable inheritance conflicts
- *type checking rules*: actual operation parameters are checked against their formal counterparts, assignments are checked for type equivalence, graph patterns and path expressions are checked against graph schemata, etc.
- *cardinality qualifier rules*: defined graph patterns should not contain obvious violations of edge cardinality constraints, deterministic graph transformations should not call nondeterministic graph transformations, ...

Figure 3.23 shows a screen dump of the PROGRES editor with the analysis tool activated. All black marks in the production `RemoveOverdueBorrower` (cf. section 3.13) indicate static semantics errors, the grey mark is a warning. The displayed error message belongs to the restriction `overdue`. It is declared to restrict sets of `Item` nodes in figure 3.7, but is applied on nodes having the type `TITLE`. The same is true for the erroneous `borrows` edge that has node `'1` as source. The production's `return` clause does also contain two errors. The first one indicates the non-existence of an attribute named `borrowerName` for the node `'1` of the class `Borrower`. The second one is caused by the attempt to assign a value to the input parameter `LostItems`.

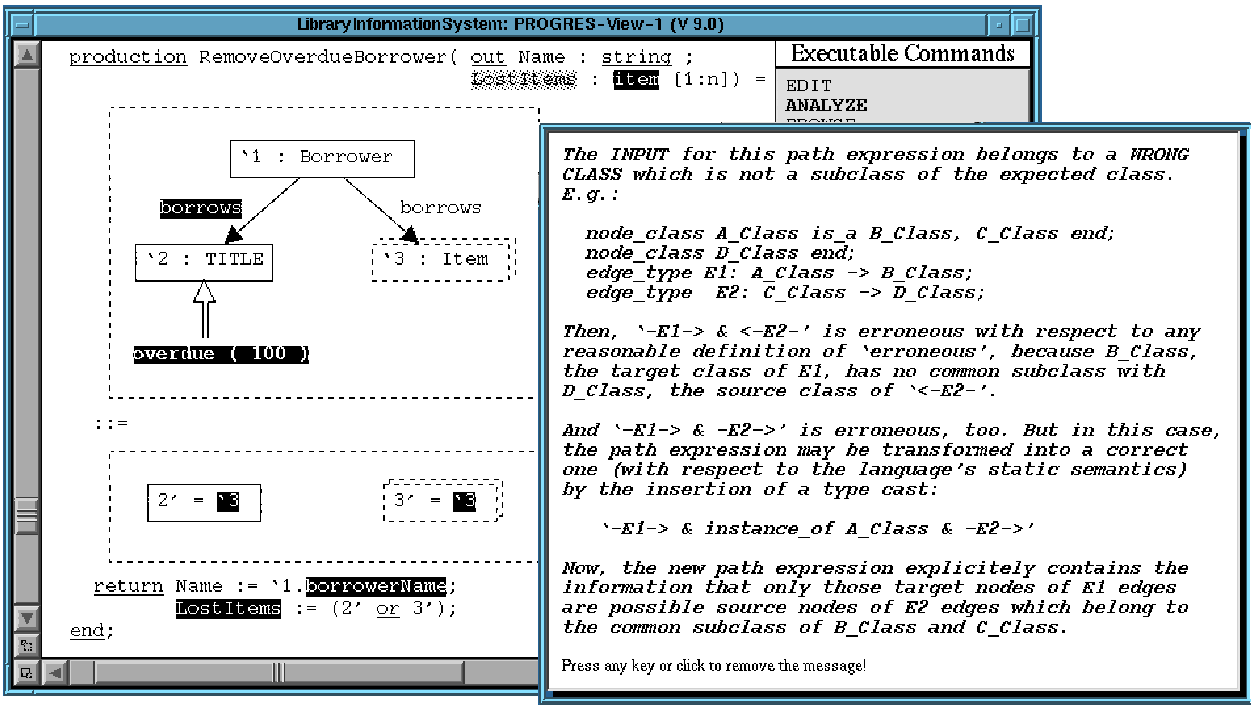


Figure 3.23: Error message analyzing a specification.

Furthermore, the two nodes in the production's right-hand side are marked. They are related to the same node in the left-hand side. Last but not least the formal parameter list contains one error and one warning. The type identifier `item` has a upper case `I` as the first letter and not a lower case `i`. The warning results from the fact that the production does not make use of the input parameter `LostItems`.

3.6.3 Executing and Debugging Specifications

The main advantage of the PROGRES environment stems from the fact that it supports more than just "drawing nice pictures". The ability to execute specifications is the most important feature for a visual programming language.

Two different possibilites exist to execute a specification:

- the *interpreter* allows to step through the specification and to make changes in the specification which are instantly executable
- the *generated prototype* provides a user-friendly interface which allows to invoke the specified operations without the interpreter overhead.

The interpreter is very useful for validating the operational behavior of the specified graph transformations directly and interactively. The *incremental compiler* translates the specification's operations into code/commands of an abstract graph machine on demand. This PROGRES graph machine code can be executed by the interpreter. At the same time C and Modula-2 code can be generated for the machine code (cf. next section). During interpretation the state of the current graph and the effects of the executed transformation can be displayed with a *graph browser*, as shown in figure 3.24. This allows the user to validate the specification.

The screen dump of the interpreter session in figure 3.24 demonstrates the debugging facilities during interpreting a specification. The visible operation in the editor is `LendItem` which we have visited before in figure 3.12. It has just been invoked with actual parameter values which are visible in the variable window on

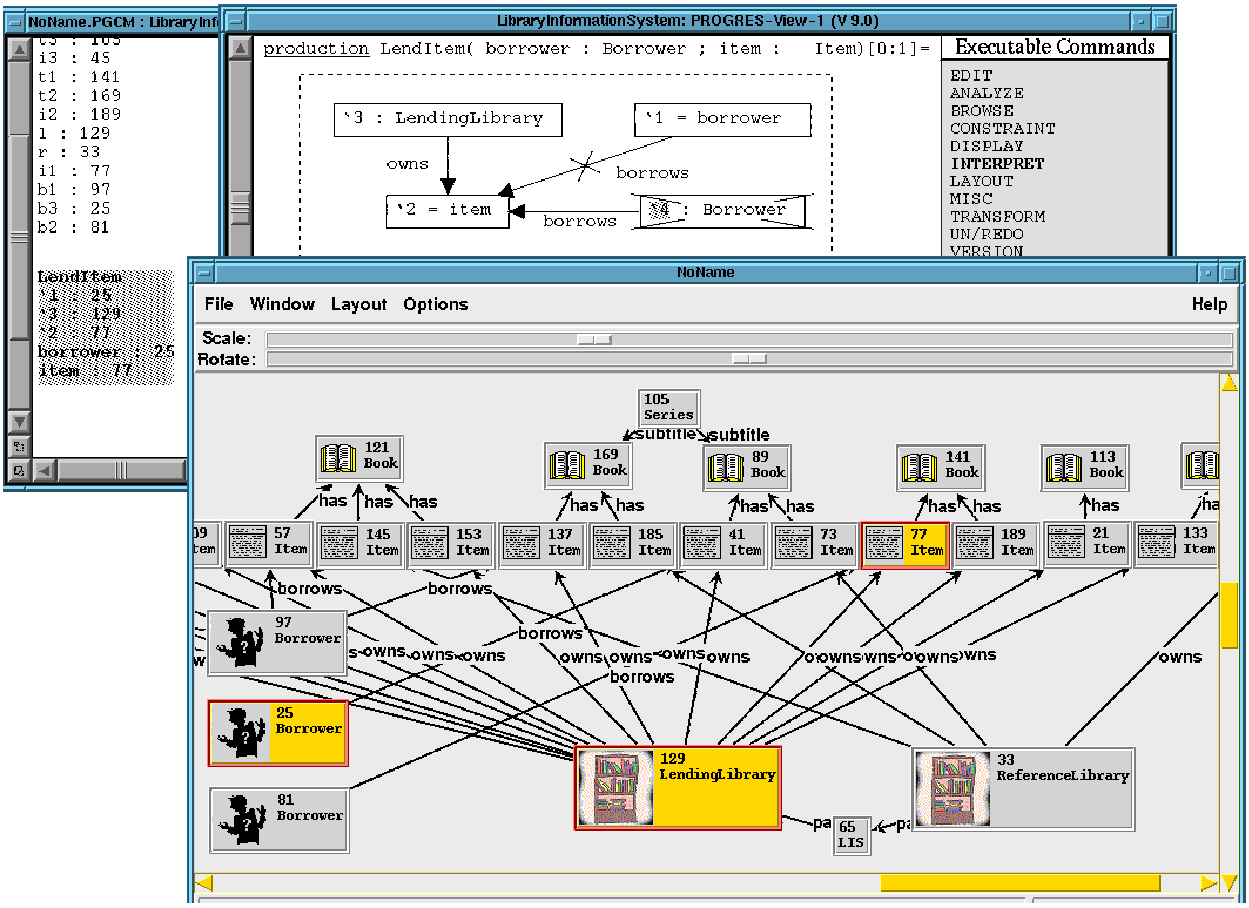


Figure 3.24: Monitoring a specification's behavior during run-time.

the left. Together with the displayed graph numbers of the graph browser we can deduce that the operation is about to fail. The current `item` with node number 47 is already borrowed by the user with node number 25. Therefore, after matching all three positive nodes in the LHS of the rule (cf. variable window), the negative node ‘4 will match the node 25 and, consequently, the application of the rule will fail.

Editor, analyzer, compiler, interpreter, and graph browser are intertwined components of the environment. If the user notices unexpected effects after applying a complex graph transformation, he may *undo&redo* single steps in order to find the erroneous operation. Together with *debugging facilities* like trace and step, the location of the error can be identified. Even without leaving the interpreter session, the specificator is able to edit and repair the specification error. The changed operation/statement can be analyzed and compiled incrementally. The interpreter is able to execute the resulting machine code immediately. That means the interrupted interpreter session can be resumed and the corrected operation can be tested directly. This allows very short turn-around times during development and error correction.

3.6.4 Prototyping

Outside the environment a PROGRES specification can be executed in a *stand-alone prototype*. It is based on the C code which is generated from the abstract code of the PROGRES abstract machine. The generated C code represents the operational behavior of the specification, which is accessible through a Tcl/Tk user interface. The user interface was built on top of the `ffgraph` library, which is one of the results of the `META Frame` project at the University of Passau.

The obvious advantage of the prototype compared to execution in the environment is that interpreting the specification is much less efficient than executing compiled C code. But, both the interpreted and the compiled code manipulate graphs which are stored in the database system GRAS, i.e. we have to pay the price for persistency of and multiple user access to manipulated graphs even for generated prototypes.

The main differences between the interpretation of a specification and the interaction with a generated prototype are (beside a speedup of factor 10):

- A prototype user is not aware of the existence of programmed graph rewriting systems as the underlying specification language.
- The execution of a graph transformation is triggered by selecting a corresponding menu entry with a meaningful name for the end user (it may be different from the specification name of the activated transaction).
- The prototype offers various means to enter needed transaction parameters (mouse selections, default values, text editor).
- View definition mechanisms may be used to tailor the visible portion of the manipulated graph such that it fulfills the needs of the end user.
- Multiple users may manipulate the same graph concurrently via prototype instances with maybe different views and different sets of commands.

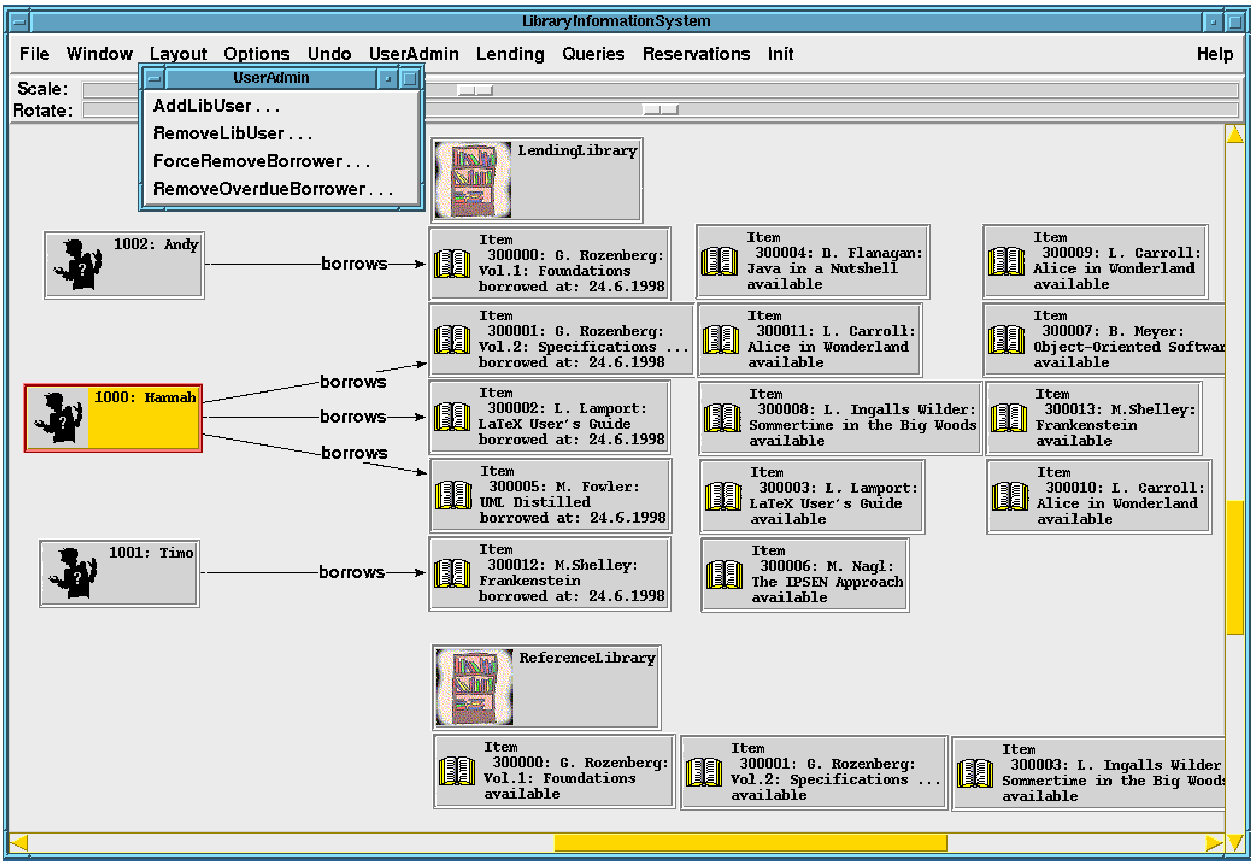


Figure 3.25: Prototype for the library information system (LIS).

A prototype offers an interface to the specification's functionality which is tailored to the end-user's needs (cf. figure 3.25. In contrast to the interpreter the prototype allows to define menus for grouping the available

operations logically. Internal operations can be hidden, i.e. made inacessible to the prototype user. This allows to provide different user groups with different access rights to a constructed graph database.

Having provided a comfortable user interface for invoking operations, it is also desirable to provide different graph views corresponding to the expectations of the user. In this context the term "view" refers to the ability to

- hide parts of the graph in its display (representation),
- compute a layout which represents the semantics of the graph, and
- create application specific graph representations ranging from diagrammatic to structured text and table representations.

The coupling between the user interface and the logical functionality in terms of graph transformations deserves a detailed explanation. First of all it has to be mentioned that graph modifications are only invoked by calling specified operations. That means it is not possible to create or delete nodes if there is not a transformation which specifies this behavior. But, compared to the interpreter session a graph transformation is encapsulated in a dialog where the actual parameters can be entered either by filling in text fields or by selecting the corresponding graph element. Any invoked transformation may have graph modifying effects which in turn have to be propagated to the interfaces of all currently active prototypes.

Figure 3.25 shows the user interface of the prototype for the user administrator. In contrast to the library administrator he must not see in every detail how a new title is inserted into the library catalogue and how key words are assigned. Instead, he is interested in the information which item is borrowed by which library user. Besides the user management operation shown in the open menu, he needs operations (offered in the remaining three menus) for fulfilling the following tasks: adding and removing users, lending out titles and taking them back, making book reservations, and querying the database for items corresponding to given authors or book titles.

3.7 Summary

The language PROGRES and its tools are the results of many years of application-oriented graph transformation (grammar) research activities with contributions from too many programmers, master thesis students, and Ph.D. students to list them all here. Nowadays, they are used at various sites for specifying and prototyping software, database, and knowledge engineering tools.

The language PROGRES may be classified as

- an object-oriented database definition language with graphical as well as textual constructs for the declaration of graph database schemata,
- a relational (logic-oriented) executable specification language with various means for the definition of derived graph properties,
- a hybrid visual/textual database query language with support for the construction of parametrized graph transformations, and
- an imperative programming language with deterministic and nondeterministic control structures for programming graph transformations.

The language's implementation uses demand-driven and incrementally working algorithms to materialize and update read-only graph views, it offers the concept of active constraints (ECA-rules) to realize updatable graph views. Backtracking is used to escape out of dead-ends of graph transformation processes, which are caused by wrong nondeterministic selections of executable graph transformation rules and their matches. The most important difference between PROGRES and most rule-oriented languages is its very elaborate type checking system. Experience shows that especially novice users need some time to remove all reported

analysis errors from their specifications. But afterwards, the chance is very high that the specification does what it is expected to do.

The PROGRES environment offers assistance for creating, analyzing (type checking), compiling, and debugging graph transformation specifications. Being an integrated set of tools with support for intertwining these activities, it combines the flexibility of interpreted languages with the safeness of compiled and statically typed languages.

A novice user finds the syntax-directed editor very helpful for creating specifications that are correct with respect to the language's context-free syntax. Later on she or he may deactivate syntax-directed editing for selected (textually represented) language constructs and use a plain text editor for entering these constructs as a whole. The built-in incrementally working analyzer offers valuable assistance for eliminating many inadvertantly made errors, which pass the checks for context-free correctness. Almost all "typos" and many kinds of usual specification errors, which have more subtle reasons, are caught this way.

The interpreter offers further means to validate an entered specification interactively and to repair incorrect implementations of graph transformations on the fly. Especially its built-in integrity constraint checking facilities and the possibility for undo and replay of execution steps is useful for tracing back erroneous graph states to the faulty piece of code. Finally, a compiler backend is available that translates specified graph transformations into C-code and generates a tcl/tk-based user interface for calling graph transformations and displaying manipulated graphs. The generated code is more or less human-readable and relies on the services of the nonstandard database management system GRAS [KSW95] and the (generic) graph editor ffgraphs. It was designed for throw away prototyping purposes.

Version 9 of the PROGRES environment is available for the operating systems Sun Solaris and Linux (alpha version) as free software. It comprises about 800 000 lines of Modula-3, C, C++, and tcl/tk code [3]. Current research activities and future development plans have a main focus on

- the refinement and implementation of the module/package concept presented here with appropriate support for specializing packages and for defining abstract graph types with updatable graph views,
- the integration of graph transformation concepts with the world of object-oriented analysis and design languages, especially with the Unified Modeling Language UML, and
- the realization of new compiler backends for rapid prototyping interactive graph-manipulating tools (in Java on top of the commercial toolkit JViews from ILOG) and object-oriented database system applications (in Java or C++ on top of the commercial OODB O2 from Ardent Software)

For further details concerning ongoing development efforts, forthcoming language and tool versions, and published papers the reader is referred to the following world wide web resource:

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

[3] PROGRES version 9 implements all explained language constructs except for the package concept discussed in section 3.5.

Chapter 4

Graph Transformation Applied To Document Image Analysis

Chapter 5

Re-Design of Legacy Applications with Graph Transformations [1]

5.1 Motivation

The success of companies and administrations heavily depends on the quality of their information systems. Existing systems are often many years old and have become very reliable over the years, but they are too inflexible to respond to changing business demands. One solution is to build a new system with the same functionality from scratch, but this is often not possible and has many unpredictable risks. Another solution is to take an existing system, to change it or parts of it with respect to the existing demands and to reuse it.

Many companies have the requirement to migrate their existing legacy applications into a distributed environment. Some parts of the existing applications have to remain on a central server because of security aspects. Other parts should be placed on local client computers which are connected by middleware [Tre96] with the server. The main task is to divide the existing applications into portions which are able to interact as client and server parts. In many cases we have to deal with COBOL programs, the data handling and technical computing should remain on a server, the user interface and local checks should be placed on client computers. Not all parts of the application can be reused. Some portions like the character-oriented user interface have to be replaced completely.

As first step of a migration process we have to reverse engineer an application to get a description on a higher level of abstraction than the source code. The acquired information is stored in form of graphs (call graphs, control flow graphs etc.). The necessary structural changes are performed on the reverse engineering graphs with the help of graph transformation rules. The transformations on the graphs are connected with source code transformations to couple changes on the abstract graph level with changes on the concrete source code level.

In the next sections we discuss the problems occurring during the re-engineering of existing applications towards a distributed one. In the following sections we present a methodology and a prototype supporting the methodology.

5.2 Methodology

To understand the problems which appear while distributing existing legacy applications we will take a short look onto the 'normal' forward engineering process for developing new distributed applications.

The first step is the development of a software architecture of the planned application. Then we have to consider which parts of the software architecture are distributed. The considerations result in the concrete cutting lines of an application. We have to determine interface definitions for the particular parts, which describe the exported resources. To use middleware conforming to the CORBA [Gro95] standard we have

[1] This chapter is an excerpt of [Cre99] written by Katja Cremer

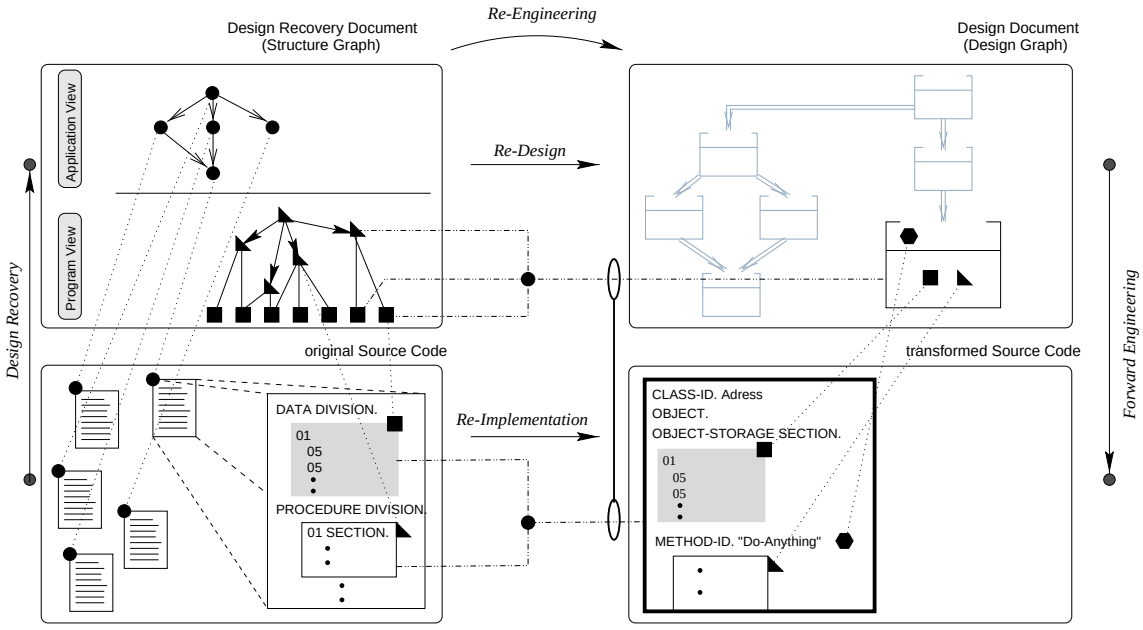


Figure 5.1: Outline of the methodology

to define the interfaces in IDL (Interface Definition Language). With the help of IDL compilers we generate skeletons in a concrete programming language. The skeletons have to be filled with an implementation realizing the exported resources. Language mappings form IDL interfaces to interfaces in a concrete programming language are defined by the Object Management Group (OMG).

Figure 5.1 shows the necessary activities to migrate a existing legacy system into a distributed environment.

There is often no structural description of a legacy application as base for a decision how to distribute an application. In this case the first necessary step towards distributed applications is a reverse engineering activity [CC90] to get a description on a higher level of abstraction than the source code level (cf. fig. 5.1 - transition from the lower left to the upper left quadrant). The result of this step is a kind of structural overview. We hopefully can identify components of the regarded system like files, programs, subprograms, data structures etc. and relations between the components like part-of, uses, call, access etc.

In most cases the structure we recognize is unsuitable for distribution because of the lack of a clean separation of different aspects of the program system. For distribution purposes the separation in different layers is suggested in the literature. For example [Gei95] proposes the three-tier model with a separation between the presentation, the technical functionality and the data handling of an application. The separated parts can use each other by interfaces.

The next step towards a distributed system is the re-engineering of the structure of a legacy system to get a separation of certain parts of the legacy application connected by interfaces (cf. fig 5.1 - transition from the upper left to the upper right quadrant). We aspire a recycling of particular parts of legacy applications by cutting them out and wrapping them with interfaces. Candidates for recycling are the parts of an application which deal with the technical functionality and the data handling. We try to create a structure of objects communicating by methods call. If we have achieved this structure we can use the objects as logical units for distribution. The concrete distribution task is described in [Rad97].

Based on the structural description we decide which parts of the application are reused in a distributed environment. For every re-engineering step on the structural level we have to define a corresponding re-engineering step on the source code level to maintain the conformance of source code with the structure description (cf. fig. 5.1 - transition from the lower left to the lower right quadrant).

We support the described methodology with a tool prototype. The prototype is mainly based on graph technology [Roz97], i.e. the reverse engineering information (cf. fig. 5.1 - upper left quadrant) is stored in

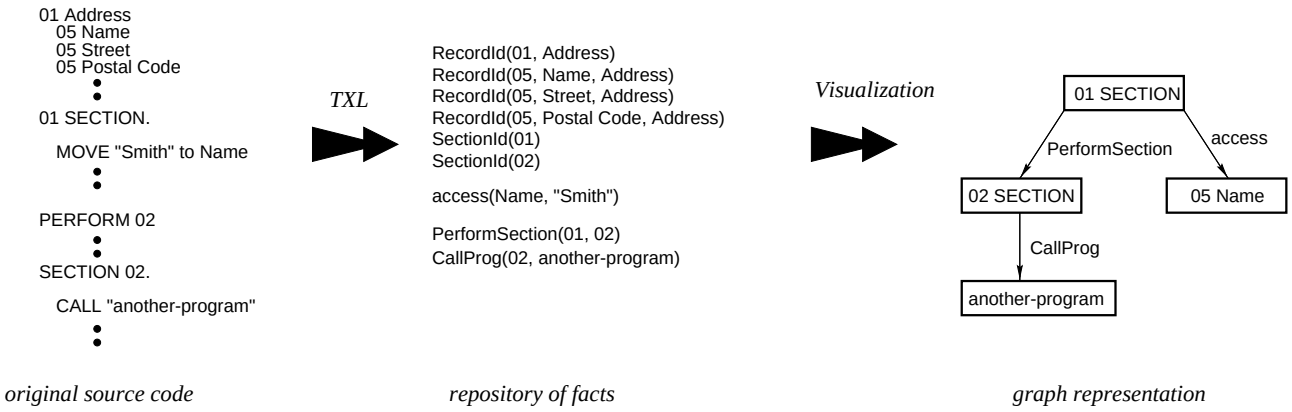


Figure 5.2: Design recovery with TXL

form of a graph. The necessary structural changes are performed on the reverse engineering graphs with the help of graph transformation rules. The several steps of the methodology and their technical realization are described in detail in the following sections.

5.3 Design Recovery

In this section tool support for the recovery of structural properties of a regarded legacy application is described. We start with plenty of source code files written in COBOL-85. What we are searching for are components and relations between the components establishing the structure of an application. Useful components are files, programs (one or more contained in a file in COBOL-85), subprograms (SECTION in COBOL-85) and data structures (contained in the DATA DIVISION), useful relationships are the calls between different programs, the subprogram calls inside a program, the data accesses and organizing relationships like a subprogram is contained in a program (for a description of COBOL see [Ste94]).

For the design recovery purposes we use the specification language TXL [CCH95]. With the help of TXL we define transformations, which extracts facts about source code artifacts. For example, TXL transformations search for specific keywords in the source code like COBOL CALL oder PERFORM statements. The transformations preserve the fact that one part of the program uses another program or subprogram in a prolog-like repository by generating facts in a dedicated target format. The left part of fig. 5.2 shows an example of a COBOL source code transformation with the help of TXL.

In the TXL distribution a design recovery specification is contained for a Pascal-like programming language. We have adapted this specification to the programming language COBOL. In the future we will enhance the specification to include external information like documentation reports or knowledge of users in the repository of facts.

5.4 Visualization of the Design Recovery Information

The repository of facts is interpreted as textual graph representation and is presented in visual form as a graph (shown at the right hand side of figure 5.2). For the visualization of facts and for the specification of the necessary structural transformations we use the specification language PROGRES [SWZ95a, SWZ95b] that is based on the concept of programmed graph transformation systems.

First of all we have to write a specification determining the behavior of the aspired tool. A PROGRES specification mainly consists of the definition of a graph schema and graph transformation rules (explained later in more detail). With the help of a generator contained in the PROGRES environment the specification is translated in C source code. A stand alone prototype arises by compiling the generated source code. The prototype has a function to read the prolog-like repository and visualizes the textual representation as graph.

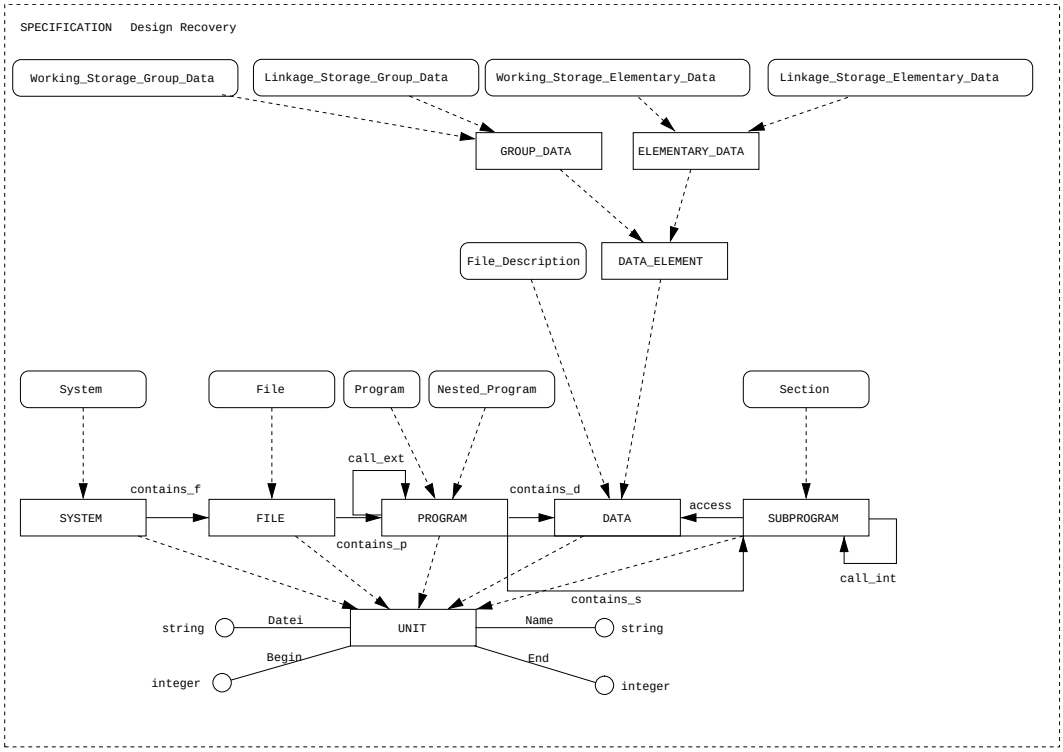


Figure 5.3: Graph schema of the design recovery document

Graphs play an important role within the area of Software Development Environments (SDEs) [Nag96]. The re-engineering prototype is a special kind of SDE supporting the software maintenance activities of an existing system. The internal documents SDEs are working on are often special kinds of graphs, for example the architecture graph of a design environment or the abstract syntax graph of a source code editor. We also model the internal structure of the design recovery document as a directed attributed graph (cf. 5.1 - upper left quadrant). Such a graph consists of labeled nodes and directed labeled edges. Attributes are needed to store additional information that need not be represented in the graph structure as, for example, the name of programs and subprograms as well as the line number of a presented artifact.

To determine the structure of the design recovery graph we have to analyze it and to identify the types of nodes, their common properties, dependencies and the relations between them. This will lead to some kind of multiple inheritance node class hierarchy called graph schema. The graph schema also includes the definition of attributes and relationships. The term graph schema is used in the same sense as the term database schema is used in database design. The acquired repository of facts is mapped onto a graph which is a concrete instance of the defined graph schema. Fig. 5.3 shows the graph schema of the design recovery document.

The root class of our inheritance hierarchy is positioned at the bottom of fig. 5.3. It has the name `UNIT`. It possesses two `string`-valued attributes namely `File` and `Name` as well as two `integer`-valued attributes `Start` and `End`. All nodes within the design recovery graph are instances of node types which belong to the class `UNIT`. As a consequence, they are owners of all of the attributes. The rest of the graph represents the different kinds of artifacts that COBOL source code may contain.

The rectangular boxes are node classes, which are from a theoretical point of view types of node types. This additional concept allow us to define common node type properties once and for all and to inherit them to node types as needed. They are connected to their super classes by means of dotted edges representing a 'is-a' relationship. The boxes with rounded corners represent node types which are connected to their uniquely defined classes by means of dashed edges representing 'type is instance of class' relationships. Solid edges between node classes represent edge type definitions, e.g, the edge type `contains_f` is a relationship

between a SYSTEM node and a set of FILE nodes, representing the fact that a program system contains a set of source code files. Circles attached to node classes represent attributes with their names above or below the connection line and their type definition nearby the circle.

The graph schema is an important part of a PROGRES specification because it defines the structure of the underlying graph. By means of generators and the PROGRES compiler, the specificator is now able to create a prototype of the modeled system. The generated prototype consists of several display components and the user can interactively work with the system and can observe the current state of the graph with a graph browser.

The following fig. 5.4 shows the generated prototype with two different graph views.

The first view a) shows nodes of the type System, File and PROGRAM. The files a system is composed of are connected by contains_f(ile)-edges with a System-node. The contains_p(rogram)-edges connect the certain programs with the files they are contained in. With the help of this view the user can get a first coarse-grained overview and a survey how many parts belong to the overall system. The main goal of this view is the identification of the programs and data inventories which belongs to an application.

The second view b) shows the internal structure of a single COBOL program. With the help of this view you can recognize certain parts of a single program. The important structural components of a single program are the data definitions (in the DATA DIVISION) and the blocks of statements (SECTIONs in the PROCEDURE DIVISION). Data accesses are represented by access-edges. SECTIONs in COBOL are a kind of subprograms without parameters. The call of SECTIONs is done by COBOL PERFORM statements, they are represented by the call_int-edges.

The two views help the maintenance engineer to get a top-down understanding of a regarded system. Beginning with an overview of the affected system components, she/he can get a more fine-grained view of the relations between the components until she/he reaches a level on which she/he may have a detailed look into the components of a single program. With the support of the generated PROGRES prototype you can view an existing system on a more abstract level than the source code documents. Generating views of the graph gives you the possibility to concentrate on certain aspects of the given system and to consider the top-down nature of the reverse engineering process.

5.5 Re-Engineering

Until now we have only considered tool support for the reverse engineering activities of the described methodology. Thus, we have reached the upper left quadrant in fig. 5.1.

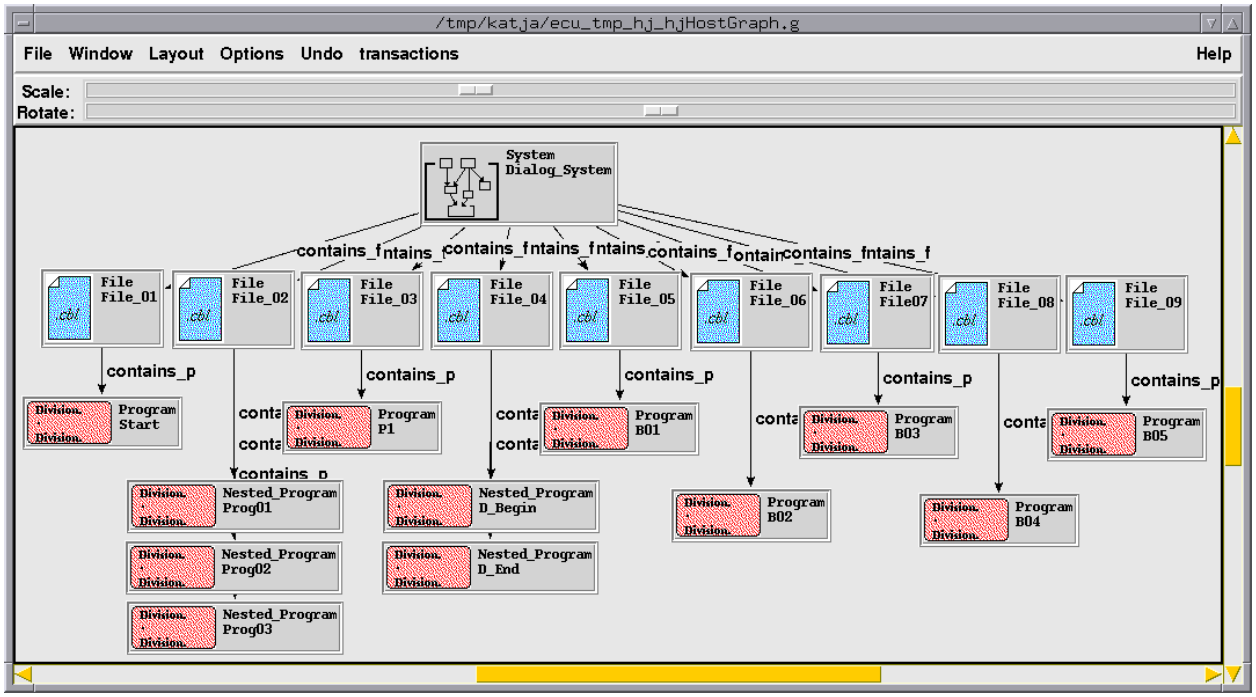
But we have the aim to migrate an existing system into a distributed environment and for this purpose we probably have to change the structure of the application. One goal of our prototype is the continuous support of the reverse and re-engineering activities. In many tools the reverse engineering process is supported well (like [Arn92, MOTU93, MWT94]), but the connection to the re-engineering activities on an abstract level misses. Reverse engineering activities supply application knowledge on the design level. On this level we want to specify the structural changes necessary for distribution purposes. But we have to connect abstract structural changes with source code transformation to couple the design with the implementation level.

In subsection 5.5.1 we describe the necessary structural changes to re-design existing applications towards distributed ones. The source code transformations are explained in subsection 5.5.2.

5.5.1 Structural Changes

The structural changes for re-designing existing applications are manifold. One of the main problem in migrating legacy applications is their monolithic character. Breaking down an application into smaller pieces and wrapping these pieces with interfaces is the main goal of the structural transformations. We describe these structural transformations in terms of graph transformation rules. The identification and specification of such rules in the PROGRES language is the main working area of the project. Most rules are used to cut some pieces out of an application and fit the separated parts together to object-like structures. In the following we present an example of an easy transformation rule.

a) View of the whole application



b) View of a single program

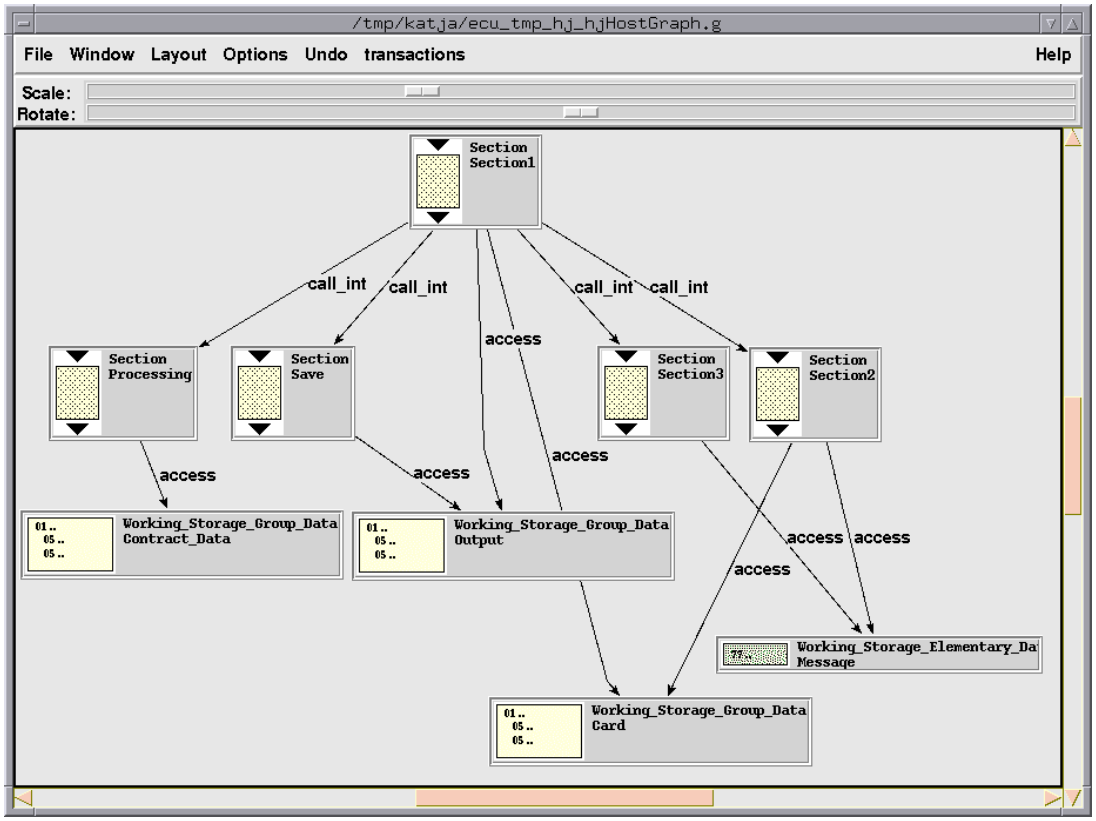


Figure 5.4: Two views of the generated PROGRES prototype

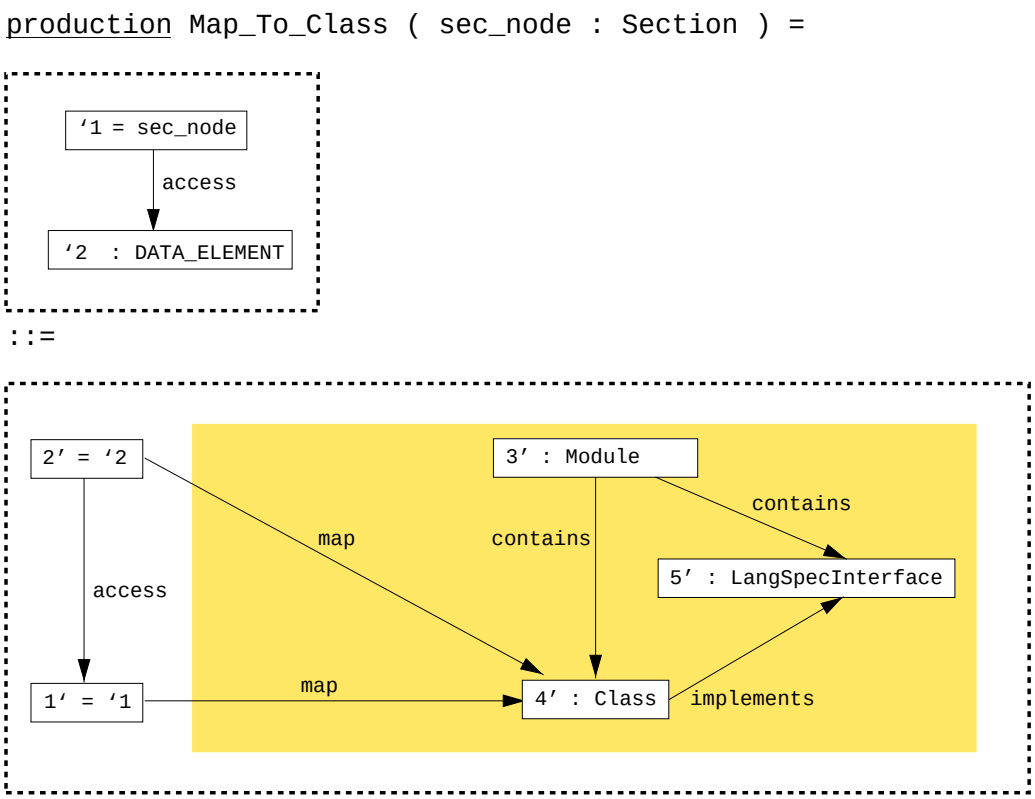


Figure 5.5: Mapping of a program to a class

In fig. 5.4 b) there is a section `Processing` which accesses the data group `Contract_Data`. There are no other sections which use this data group. In the object-based sense we have to decide between two possibilities. On the one hand we could decide that the data group is an important data structure for the application and should be used as abstract data object. In this case the data group is separated and we have to define methods which handle the access onto the data structure.

On the other hand we could group the two pieces together interpreting the data group as attributes and the procedural part as methods of a class. Fig. 5.5 shows the corresponding graph transformation rule.

On the left hand side of fig. 5.5 (the part above the '::=') we define the graph structure we are searching for (a subprogram (in COBOL `SECTIONs`) accessing a `DATA_ELEMENT`) and we want to reuse in an object-based scenario. Because of the inheritance hierarchy different concrete types of data elements are possible (cf. fig. 5.3).

On the right hand side we define the replacing graph structure. There are some node and edge types (grey underlayed) which you don't find in the schema definition of fig. 5.3. These are part of an architectural definition language (ADL), which we use to define the desired object based structure. An example of a design document using such a kind of notation you find in fig. 5.1 in the upper right quadrant.

The notation `x' = 'x` stands for the identical replacement of nodes. This means that the original nodes and edges of the left hand side remains, additionally a `Class`-node is introduced and the original nodes (`1` and `2`) are mapped onto it. This mapping has the semantics of reusing the original parts in a new object-based structure.

Furthermore a `Module`-node and a `LangSpecInterface`-node are added. The `Module`-node is connected via `contains`-edges with the `Class`- and `LangSpecInterface`-node. This expresses that a module is composed of two parts. In fig. 5.1 modules are symbolized by boxes, where the upper open part of a box is the interface and the closed lower part represent the implementation of the interface. This relation between the interface and its realization is expressed by the `implements`-edge in fig 5.1.

This graph transformation rule realizes the reengineering on the design level by assigning existing parts of an application to components of an architecture description language expressing the desired structure.

Fig. 5.5 shows the specification of the transformation rules. With the help of the PROGRES source code generator you can create executable code. The transformations can be invoked in the PROGRES prototype environment on selected graph nodes.

5.5.2 Source Code Alterations

Every node in the graph points to some piece of source code. In fig. 5.3 the root class of the inheritance hierarchy possesses two `string`-valued attributes namely `File` and `Name` as well as two `integer`-valued attributes `Start` and `End`. These attributes pick up the name of the COBOL file artifacts are contained in and the start and end position of artifacts in a file as well as the name of artifacts. Every structural transformation specified by a graph transformation rule must have a corresponding source code transformation to incorporate the re-design into the implementation level. The source code transformations are under development. We will implement source code transformation rules from ANSI COBOL-85 to Object COBOL. For example you can reuse parts of the `Working-Storage-Section` as attribute definition in the `OBJECT`-part of a class definition. The main goal of the source code alterations is the suitability for the use of the CORBA COBOL language mapping.

The transformations of the source code are also defined with the help of TXL. The coupling with a graph transformation rule is done by external C functions which are offered in the PROGRES language. C functions call the necessary TXL transformations rules responsible for the source code transformation. Every graph transformation rule has a corresponding textual source code transformation and every application of a graph transformation rule entails a source code transformations by calling external C functions.

5.6 Summary

In this paper we have presented a tool prototype for the re-design of existing applications. The aim of the approach is the migration of existing applications into a distributed environment of components communicating by middleware products conforming to the CORBA Standard.

The underlying methodology provides as first step the extensive reverse engineering of the regarded application. This is done with the help of the specification language TXL. The recovered design information is visualized as a special type of graph. The type of the design recovery graph is defined in the PROGRES language and the acquired prolog-like facts are mapped onto graphs of this type. The design recovery graph presents information on higher level of abstraction than the source code. With the help of graph views the maintenance engineer can get a understanding of an application.

The graph is not only used for reverse engineering purposes, but also as base for the re-design of an application. The re-design transformations are defined as graph transformation rules. One problem of existing applications is their monolithic character. The re-design tries to break down an application into smaller pieces and put the fitting parts together in an object-like structure.

Every transformation on the structural level must have corresponding transformations on the source code level. The transformations are performed such that CORBA can be used for distributing the resulting program.

We are working on a wide spectrum of transformations on the design level with corresponding transformation rules on the source code level. Future work is done in the area of flexibility for other programming languages and other re-design goals than distribution.

Chapter 6

Modelling a Visual Language with PROGRES[1]

Graph grammars are often used for the definition of the syntax and semantics of visual languages like e.g. in DiaGen [MV95]. Beyond that there are a still growing number of visual languages that rely directly on the graph rewriting paradigm, such as Ludwig2 [Pfe95], GOOD [PBA+92] or PROGRES [SWZ95a].

The development of a visual language with graph transformations presumes that this language has a diagrammatic representation or its abstract syntax can be represented by a graph. A popular class of visual languages, dataflow languages, fulfils these requirements. Dataflow languages can be understood as a sort of functional programming languages with a visual representation of the functions. One of the successful visual dataflow languages is Prograph [CGP95]. Prograph is an object-oriented programming language which is intended for commercial application development. Dataflow languages in general have a big commercial potential. The success of Prograph and also LabVIEW [VW86] which is a visual dataflow language used in laboratories in the industry as well as in the academic context shows that.

A visual graph-based language can be described by using a graph grammar. From this graph grammar a syntax-directed editor, a parser etc. can be generated. However, in this chapter we will concentrate on the modelling process of a visual dataflow language with a graph transformation system which describes the operations of a programming environment for visual (dataflow) languages like editing, analyzing and executing programs specified in this visual language. We will show how to specify those operations and how to generate a prototype of such a programming environment.

For this reason we introduce a small example language **HotVla**[2] which is a simple dataflow language. For the purpose of this tutorial other dataflow languages such as Show & Tell [KCM90], Prograph [CGP95], or BDL [Sch97a] are too complex. However, it is also possible to model (parts of) these languages with PROGRES.

The next section describes the syntax of HotVla briefly. The scheme of the graph describes the interdependencies of language constructs and expresses some integrity conditions regarding well-formedness of a specified program in this language as well. Section 6.2 presents some productions specified in PROGRES that describe editing operations in the generated prototype. Section 6.3 points out an idea how to implement a type inference system for visual dataflow languages with PROGRES. Finally, section 6.4 shows how a HotVla program will be executed using the generated prototype.

6.1 Concepts of Modelling a Visual Language

In this section we introduce the concepts of modelling a visual programming language with PROGRES. For this purpose we introduce a language called **HotVla**. We will show how this language has been defined in

[1] Excerpts from a forthcoming technical report written by Manfred Münch.

[2] HotVla is an acronym for Higher Order Typed Visual LAnguage

PROGRES. More precisely, PROGRESwas used to specify the operations of an editing tool, an analyzer, an animation tool and to generate a programming environment prototype for HotVla.

6.1.1 The HotVla language

One of the main reasons for the lack of a model for a visual programming language is the variety of approaches researchers have developed. Within similar areas there are still differences in the approach taken to develop a language. Most languages are only special purpose languages which makes it even more difficult to compare one language to another and find common elements.

To be able to develop a basic language which is simple, yet powerful enough to have the same expressiveness as e.g. primitive functional languages we have decided to base our language on already existing calculi. The central theory behind our dataflow language HotVla is the λ-calculus, enriched by the concept of higher order dataflows (see [Tyu91]). However, details about the theory of that language are not discussed here.

The HotVla language is designed for being the core of a visual dataflow language definition kit. Existing dataflow languages should be definable by or reducable to HotVla programs. To develop mechanisms and to build up a library for this purpose is left to future work. In this chapter the HotVla language serves as an example which is not too complex.

A main goal for the language design was to keep the number of different language constructs minimal. Because of the extension of common dataflow concepts (where the values are either of a primitive type or a tuple of primitive types) to a higher order dataflow concept, i.e. dataflows may also have functions as values, it is possible to define a general purpose language with only two control elements besides a set of atomic functions like '+', '−', '<', '=' etc. and constant values.

The next subsection demonstrates how we specify the syntax of HotVla in PROGRES. After that we present how to build a very simple editor for our language.

6.1.2 The HotVla Syntax

The syntax of HotVla is kept minimal. Next to the already mentioned arithmetic operations we have only two control structures. The first one is a simulation of an if-then-else-construct, the second is a function call element. For a proper data handling we need an operator *pair* which constructs a tuple out of two (ordered) inputs, and the functions fst and snd which extract the values again. With that we can restrict ourselves to unary functions only. Beyond that we are already able to specify any mathematical function.

The first control element we introduce is the function call. This enables the language to express repetitions by using recursion. We use the notation for a function call given in fig. 6.1. In this example we call the function p with an argument carried by the dataflow a. The result of that call will be carried by the dataflow b. Informally, we can write $b = p(a)$. How we model functions in HotVla will be described later in this subsection.

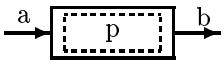


Figure 6.1: Notation of a call of function p with input a, producing output b

All arithmetic functions are actually nothing else than a function call with a tuple as argument. However, we have added some syntactic sugar so that it is allowed to write an expression like shown in fig. 6.2 as long as they are commutative. Furthermore we omit the dotted lines for those standard functions.

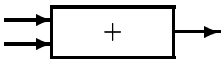


Figure 6.2: Intuitive notation of an arithmetic function as long as the order of the input is insignificant

The second control element we introduce is the simulated if-then-else-construct, which we call a *Switch*. The appearance of that node is shown in fig.6.3. Although the three dataflows are labeled we do not differ

between any kind of dataflow. The labels can be understood as the names of the ports (the two input dataflows of a *Switch*-node are not interchangable). The dataflow leading to port *Cmp* carries a boolean value which is used for deciding which function will be called, *tc* (like *then-case*) if that dataflow carries the *True*-value, *ec* (like *else-case*) otherwise. The value of the dataflow leading to port *Data* is used as an argument to either of these functions. The result of that conditional function call will be carried by the outgoing dataflow.

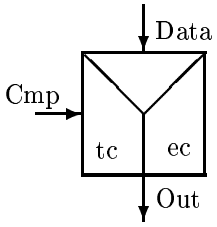


Figure 6.3: Switch node: modelling of an if-then-else-construct

Function declarations consist of two nodes: the *FunctionDecl* node defines the name of the function and serves as "port" for the value given to the function. The *Output* node serves as "sink" for the data to be returned to the calling function. Fig. 6.4 shows an example of a HotVla program implementing the factorial function. (Note that the ports of the *Cmp* and the − operation are labeled 1 and 2 for the first and the second operator, *CF* is the port for the compare function). The *Cmp* function is an example for the aspect of HotVla being a higher order dataflow language. Those ideas can be extended even further but this is beyond the scope of this tutorial.

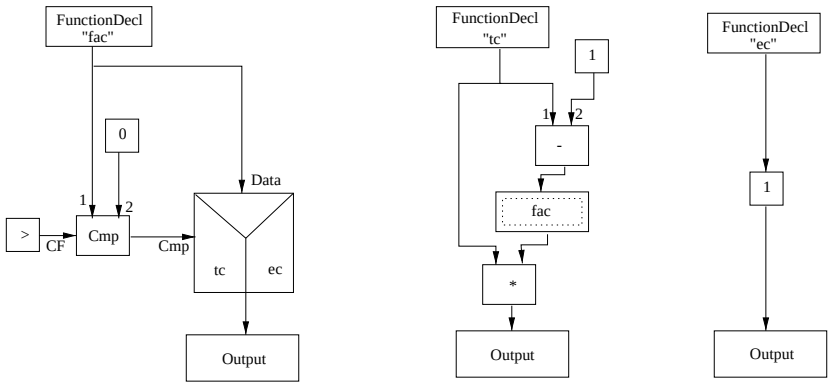


Figure 6.4: The factorial function modelled in HotVla

Until now we have specified the syntax of the HotVla language informally. A formal definition of HotVla's syntax is not provided in the form of a graph grammar but (in our example) in the form of a PROGRES graph schema definition with associated integrity constraints. First, we define a node class with all common properties any language construct should have. We have called this class $HotVla\_ELEM$ (see fig. 6.5). Classes are represented by rectangulars. Now all refinements of the class $HotVla\_ELEM$ can inherit the properties like *Representation* of type *string* which will contain some text describing the node, e.g. $'+'$ for one of the arithmetic operators. The refines-relationship is modelled by the dotted arrow, e.g. from $DECL$ to $HotVla\_ELEM$. The rectangulars with round edges represent node types, the actual instances of node classes thus. Some of these node types and classes define additional attributes (although for the sake of readability only a few are shown in fig. 6.5). The node type *FunctionDecl* e.g. is an instance of the node class $DECL$ and defines the string attribute $FunctionName$ in addition to the already inherited attributes. This attribute stores the name of the function which can be called by the $Call$ construct. For this reason the string attribute $Function$ must contain the function's name, of course.

The representation of the *Switch*-node in the scheme is slightly more abstract than shown in fig. 6.3. We have to use the word "Switch" for the representation since PROGRES does not support to define the language elements graphically as we have chosen in fig. 6.3.

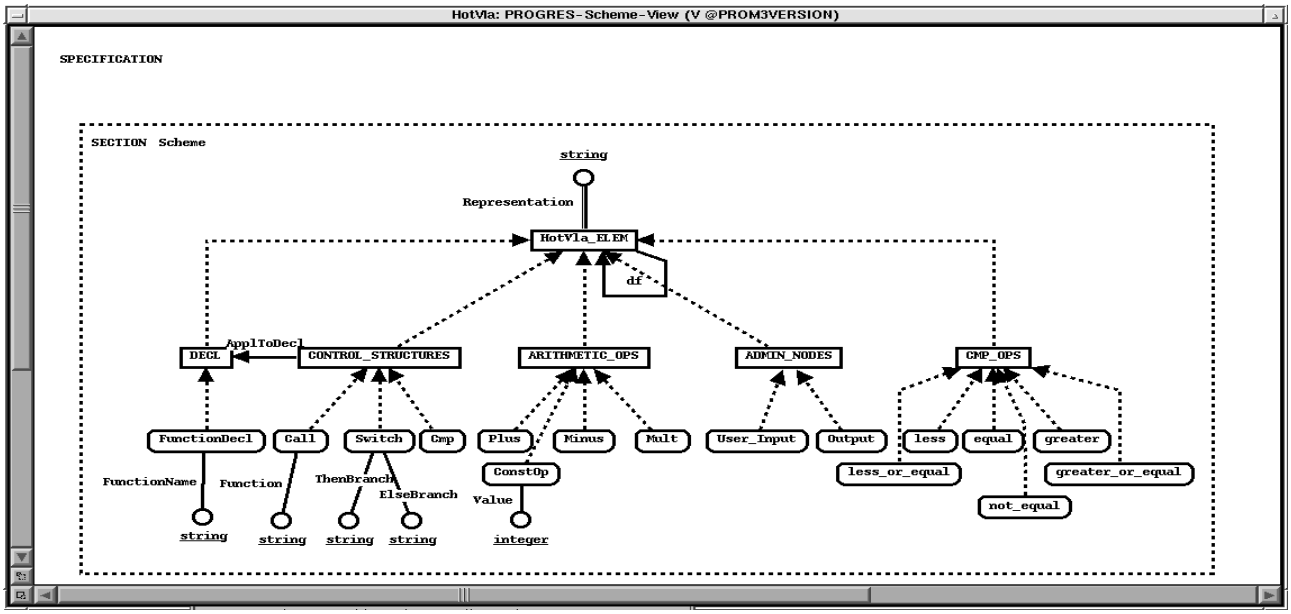


Figure 6.5: A part of the language definition of HotVla

Furthermore we have defined two edge types. The edge *df* may connect any node with any other node and will be a general dataflow in our language. The edge *ApplToDecl* connects a *Call* node or a *Switch* node with a *FunctionDecl* node. A required integrity constraint is to make sure that the function calls of either the *Call*-nodes or the *Switch*-nodes are indeed declared by the function declaration the *ApplToDecl*-edge points to. We will need this for type checking in section 6.3.

Dataflows can be modelled in two different ways. The simple solution which we have chosen is depicted in fig. 6.5. A dataflow is modelled by an edge which can connect two arbitrary nodes, i.e. language constructs, of our language. Eventually some constraints are needed to ensure the integrity of a program (an *Output* node may not have any outgoing dataflow etc.). The second possibility is to define a dataflow by an extra node which is connected to the actual source and target by the use of derived attributes. Then we can also model data consuming and non-consuming functions. For the sake of simplicity we have not chosen this second solution although it is the more powerful one.

However, this scheme definition contains some more problems. The *ApplToDecl* edge e.g. may not connect a *Cmp* node with any instance of the *DECL* node class. The scheme allows this connection but it does not make any sense. Beyond that it is allowed to have more than two input dataflow at the *Switch* node although this does not make any sense either. This can be prevented in three different ways:

- integrity conditions built into the scheme make sure that those erronous situations do not occur
- appropriate editing operations prevent an erronous specification (see section 6.2)
- additional analyses make sure that a specified program fulfils the syntactical correctness.

The next section describes how we can specify editor operations in PROGRES. We will also demonstrate how to implement context-sensitive operations to prevent erroneous editing. Finally we show an example of the editing process in the generated prototype of the programming environment.

6.2 Generating a HotVla Editor

After having shown how to define the structure of a visual language with PROGRES we demonstrate how to build an editor for this language. This editor does not only create one node after the other which have to be combined by the user manually. We show how to develop an editor that has a bit more intelligence. Having a look at our language it is straightforward that a function must have the node *FunctionDecl* and the node *Output*. If we specify a PROGRES production creating these two nodes and connecting them by a dataflow edge *df* we can make other productions dependent on the created context. That means that another node can only be inserted if it finds the context it needs. Fig. 6.6 shows examples of two productions implementing the editor.

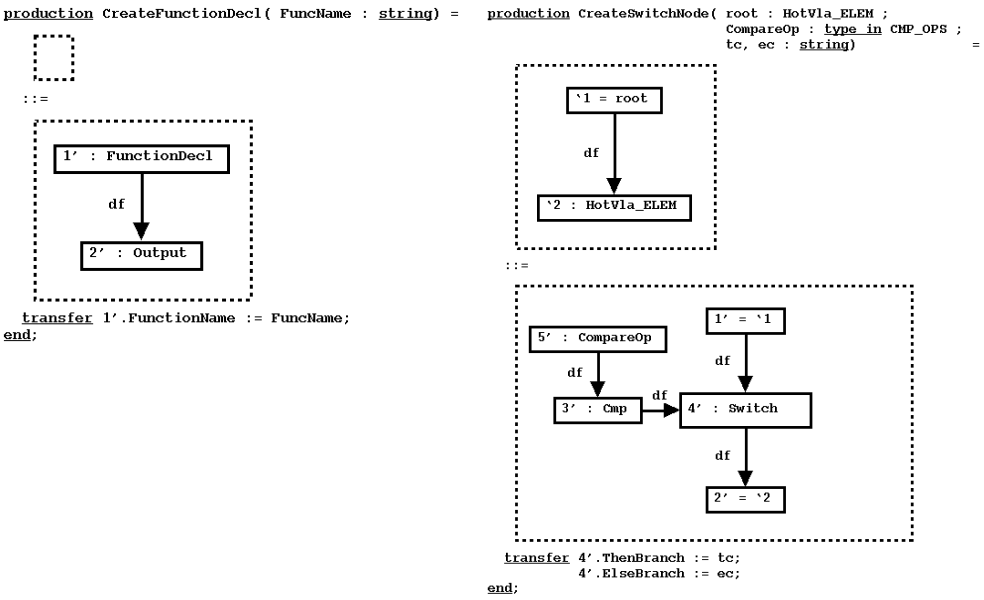


Figure 6.6: Productions implementing *CreateFunctionDecl* and *CreateSwitchNode*

The usage of these productions in the final generated programming environment is depicted in fig. 6.7. In the shown situation the context for applying *CreateSwitchNode* has already been created and the operation is called. The user is asked to provide the system with some necessary information like the father-node where the input data comes from (*root*), the compare operation associated with the *Switch* and the function names for the *ThenBranch* and *ElseBranch*.

The other productions which are needed to realize an editor for our language look very similar. We do not need any more complicated operations. These productions cover already some context-sensitive analyses. In the next chapter we will discuss other analyses we can apply to a specified program. Beyond that we will present some ideas how to realize a simple type checker with PROGRES for the HotVla language.

6.3 An Analysing Mechanism for HotVla

In fig. 6.6 in section 6.2 we have seen that many context-sensitive analyses can be omitted if the editor is specified with a little knowledge about the language. If this is not the case (e.g. the user wants to be able to leave a program in an inconsistent state temporarily) it is not very difficult to specify some tests in PROGRES which will check the syntactical correctness of a HotVla program as an alternative to parsing the program based on the grammar. These tests look very much similar to the productions shown in fig. 6.6. Therefore we omit them here.

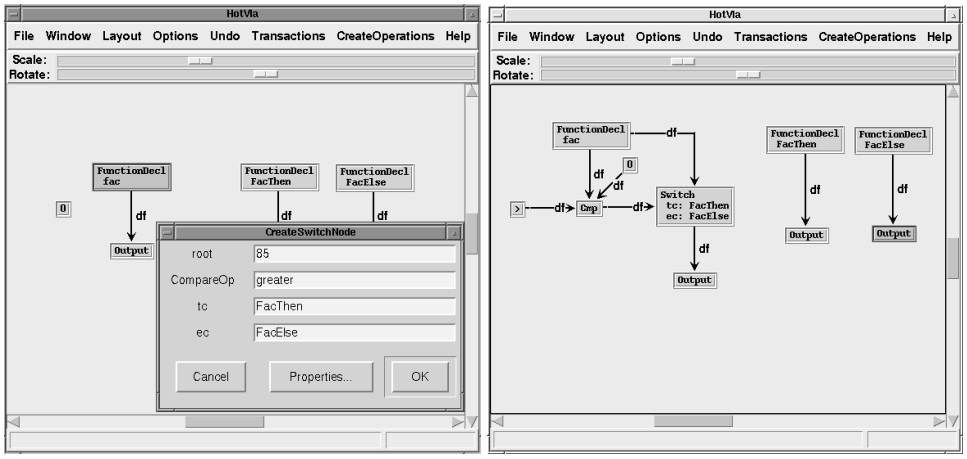


Figure 6.7: Usage of the productions shown in fig. 6.6 in the final generated programming environment. Left: Input of the necessary parameters, Right: After the insertion of a *Swich*-node

An interesting check an analyzer could perform on a specified HotVlaprogram is a type check. The idea is to add some type information to every node in the way it is known from e.g. functional languages. With these information it is possible to implement a type inference system.

E.g. arithmetic operations shown in fig. 6.5 will all get the type e.g. $Plus :: int \rightarrow int \rightarrow int$. This type information will be modelled by a set of nodes describing the in- and output data types. The edges at ports in_t_info and out_t_info (in fig. 6.8 the port identifiers are annotated at the edges) connect the type information nodes to the function nodes. Generic types like $\alpha, \beta, \gamma, ...$ (or $*, **, * * *, ...$) are also allowed. Then we try to match the types of two successing functions as follows: from every predecessing output data type we draw an edge to every input data type. Fig. 6.8 illustrates that. We can treat function calls or switches, which call one of two functions implicitly, equally. Their input and output data type is determined by the 1-context. However, the principle of type matching is the same.

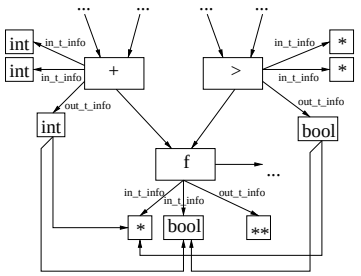


Figure 6.8: Every output data type is connected to every input data type of the following function

A simple production can identify all edges with similar types at both ends and delete this edge including both type nodes. The other type information nodes should be embedded properly so that each node has still a syntactically correct type information chain. Fig. 6.9 shows such a production inserting the edges for matching types and another production that eliminates these matching types.

Finally only non-matching pairs of type information are left in the enriched program. There are two possible cases:

- At least one of the type information is a generic data type. Then a unification of these two will do and we can delete them.
- Both data types cannot be unified. Then we have a type clash which indicates a wrongly typed program.

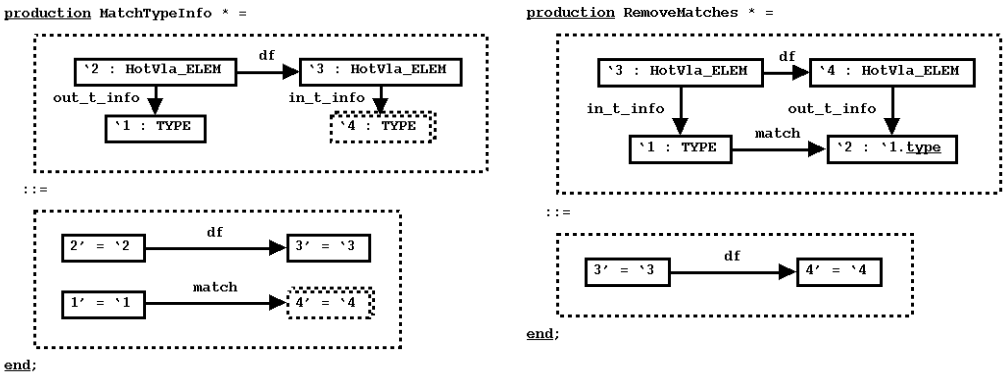


Figure 6.9: Production that inserts the *match* edges (left) and the production that eliminates these matching types (right)

The only problem left is if it is possible (or allowed) to unify two different generic data types or if this is a type clash. We are currently investigating on this topic.

6.4 Executing HotVla in the Generated Prototype

After editing and analysing a program in our visual language we also want to execute the specified program. In this section we show how the execution machinery of our HotVla language works. It is possible to generate a rapid prototype of the specified language to have a standalone application, i.e. editor, analyzer, and animator, of the language. We will demonstrate the animation with the help of this generated prototype. However, the process of generating this application is not the topic of this section.

The execution of HotVla is demand driven, i.e. beginning at the *Output* node the machine tries to compute this node. For this reason it computes its predecessors. If they have no computed value yet this process will continue recursively until all values are available. Analyses of the specified program could make sure that there is always a possible way of execution. The termination problem cannot be solved, of course.

The animation machinery is realized as an interpreter with a program stack. All values of functions are stored in attributes to the nodes. The dataflows connect the functions but do not have values themselves. A way of realizing edges carrying values is to introduce an intermediate node between the source- and target-node of the edge. This subsidiary node has the only task to store the value of the dataflow. The former source- and target-node of the original edge are connected to this intermediate node. With that it is also possible to realize a data-consuming or non-consuming machinery, depending on whether the node will be deleted when the value has been read or not.

The semantics of the arithmetic and the compare operations is straightforward. With that the way of executing those rather primitive functions is very clear. The administrative nodes in our language call externally defined functions for user interaction. *User_Input* e.g. calls a C-function that prompts for an input. Furthermore the *DECL* nodes need no explanation because they do not have any operational semantics.

The interesting part of the language definition is the part defining control elements. *Cmp* e.g. takes three inputs, i.e. two values of the same type and a compare function valid for this type, and combines them. The result is of type boolean.

A little bit more complex is the semantics for the *Call* node. To simulate the way an operational semantics is defined we have decided that the execution does not perform a *jump* to the function but unrolls the definition and replaces the *Call* node by an instance of the function. For this purpose, every node of the called function will be duplicated and properly embedded so that the function is copied. We store the nodes connected to the *FunctionDecl*-node of that copy and connect these to the predecessor of the original *Call* node. Then the *FunctionDecl*-node of the copied function can be deleted. The *Output*-node will be deleted in a similar way.

The semantics of the *Switch*-node is very similar. The only difference is that the node itself will not be deleted like the *Call*-node but stays in the graph (we are aware that this is not consistent and might be changed in future versions of HotVla). The copied function, which was selected by the *Switch* previously, will be embedded behind the *Switch*-node and the input value of the *Switch*-node will be led through to the embedded function.

Fig. 6.10 shows the execution of the factorial function in our generated programming environment. The left picture is divided into two parts: the three graphs of the function definition of the factorial function are shown in the left part of the figure, the program execution is the right graph. There we have already executed the *Switch*-node and inserted the *FacThen*-function behind that node. The next step is to execute the function call. The result is shown in the right window.

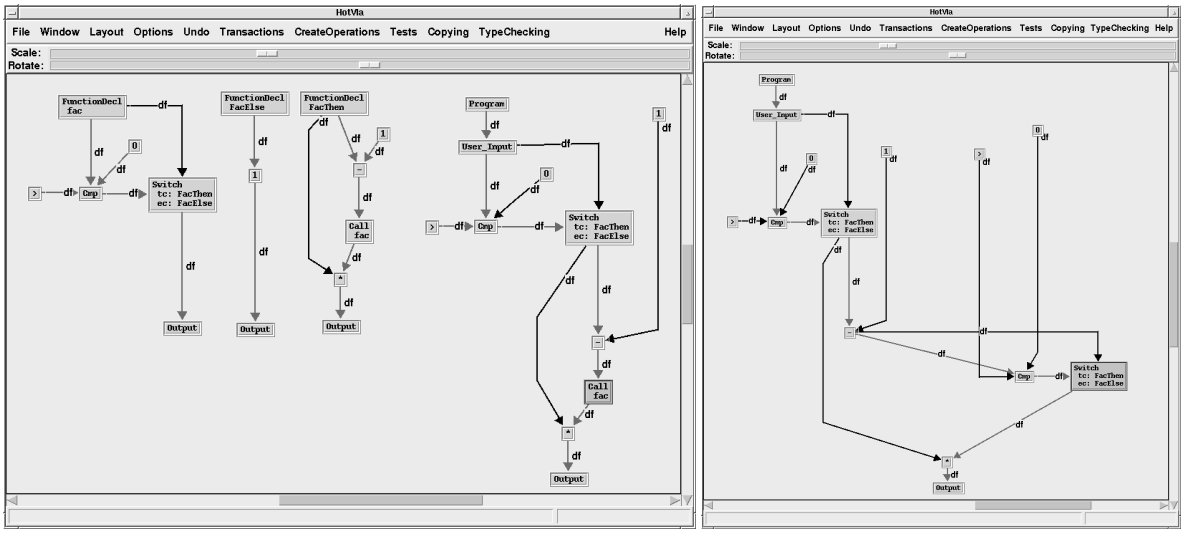


Figure 6.10: Left: The factorial function after executing the *Switch*-node (rightmost graph). The three leftmost graphs define the factorial function. Right: The function call has been executed and another copy of the fac-function has been inserted.

By implementing an animation tool with PROGRES in the way demonstrated here it is not difficult anymore to deduct the operational semantics of the specified visual language. The machinery simulates already the unrolling of function calls (which may also be recursive) and works demand driven, which can also be mapped to the bottom-up development process of an operational semantics to a specified program.

6.5 Summary

In this chapter we have shown how to develop a visual language with PROGRES. First we have defined the structure of the language with the help of the visual object-based PROGRES Scheme Editor. The next section has demonstrated how a context-sensitive editor for a visual language can be built that saves a lot of analyses on a specified program regarding the syntactical correctness. Nevertheless, additional analyses can be specified. We have presented a possible implementation of a type inference system for our simple dataflow language. Finally we have shown that PROGRES not only gives its user the opportunity to build an interpreter for his language.

The example throughout this chapter was a first version of the HotVla language. This language is still under development. Papers about the theoretical background of HotVla are on their way. We have designed this language as the core of a development kit for visual dataflow languages. In the future we want to define a library of generic modules and specialisations of these to define several classes of visual languages.

Chapter 7

Conclusion

Bibliography

[AE94] Marc Andries and Gregor Engels. Syntax and semantics of hybrid database languages. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 19–36, 1994.

[AEH+96] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. Technical Report Informatik 7/96, Universität Bremen, Germany, 1996.

[Arn92] Robert S. Arnold, editor. *Software Reengineering*. IEEE Computer Society Press, 1992.

[Aßm96] Uwe Aßmann. On edge addition rewrite systems and their relevance to program analysis. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 1996.

[Bei95] C. Beierle. Concepts, Implementation, and Applications of a Typed Logic Programming Language. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 5. Elseview Science Publ., 1995.

[BGL95] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments*. Manning Publications Co., Greenwich, 1995.

[BL93] B. Bell and C. Lewis. Chemtrains: A language for creating behaving pictures. In *[VL93]*, pages 188–195, 1993.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings Series in Object-Oriented Software Engineering. Benjamin Cummings, Redwood City, CA, 1994.

[BS86] R. Bahlke and G. Snelting. The psg system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[Bun79] Horst Bunke. Programmed graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 155–166, 1979.

[CAB94] Derek Coleman, Patrick Arnold, and Stephanie Bodoff. *Object-Oriented Development: The Fusion Method*. Prentice Hall, London, 1994.

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 12(1):13–17, 1990.

[CCH95] J.R. Cordy, I.H. Carmichael, and R. Halliday. *The TXL Programming Language (Version 8)*. Legasys Corp., Kingston, 1995.

[CD94] S. Cook and J. Daniels. *Designing Object Systems with Syntropy.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[CER79] V. Claus, H. Ehrig, and G. Rozenberg, editors. *Proc. Int. Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, Berlin, 1979. Springer Verlag.

[CGP95] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph. In *[BGL95]*, pages 45–66. 1995.

[Chr68] C. Christensen. An example of the manipulation of directed graphs in the ambit/g programming language. In *Interactive Systems for Experimental and Applied Mathematics*, New York, 1968. Academic Press.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation – part i: Basic concepts and double pushout approach. In *chapter 3 [Roz97]*, pages 163–246. 1997.

[Cou97] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *chapter 5 in [Roz97]*, pages 313–400. 1997.

[Cre98] Katja Cremer. A tool supporting the re-design of legacy applications. In Paolo Nesi and Franz Lehner, editors, *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pages 142–148. IEEE Computer Society Press, 1998.

[Cre99] Katja Cremer. *Reverse and Reengineering Tools based on Graph Transformations Systems.* PhD thesis, Department of Computer Science III at the Aachen University of Technology, 1999. (will appear in 1999, in german).

[DFS74] E. Denert, R. Franck, and W. Streng. Plan2d - towards a two-dimensional programming language. volume 26 of *Lecture Notes in Computer Science*, pages 202–213. Springer Verlag, 1974.

[Dil92] A.Z. Diller. *Z: An Introduction to Formal Methods.* John Wiley, New York, 1992.

[DK76] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, 1976.

[DW96] Desmond D'Souza and Alan Wills. Extending Fusion: Practical rigor and refinement. In Ruth Malan, Reed Letsinger, and Derek Coleman, editors, *Object-Oriented Development at Work*, pages 314–359. Prentice Hall, Englewood Cliffs, NJ, 1996.

[EE96] Hartmut Ehrig and Gregor Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 1073 of *Lecture Notes in Computer Science*, pages 137–154. Springer, 1996.

[Ehr87] Hartmut Ehrig. Tutorial introduction to the algebraic approach of graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 3–14, 1987.

[EHTE97] Gregor Engels, Reiko Heckel, Gabriele Taentzer, and Hartmut Ehrig. A combined reference model and view-based approach to system specification. In *[?]*, pages 457–477, 1997.

[EP98] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit.* John Wiley, New York, 1998.

[EPS73] Hartmut Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled.* Addison Wesley, New York, 1997.

[Fur91] G. W. Furnas. New graphical reasoning models for understanding graphical interfaces. In *Proc. of Conf. on Human Factors in Computer Systems - CHI'91*, pages 71–78, New York, 1991. ACM Press.

[Gei95] Kurt Geihs. *Client/Server Systems: Foundation and Architecture.* International Thomson Publishing, 1995. (in german).

[GGN91] Herbert Göttler, Joachim Günther, and Georg Nieskens. Use graph grammars to design CAD-systems! In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 396–410, 1991.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specifications.* Springer Verlag, Berlin, 1993.

[Gli90] Ephraim P. Glinert, editor. *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Göt83] Herbert Göttler. Attributed graph grammars for graphics. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 130–142, 1983.

[Göt88] Herbert Göttler. *Graphgrammatiken in der Softwaretechnik*, volume 178 of *Informatik-Fachberichte.* Springer-Verlag, Berlin, 1988.

[Gro85] CIP Language Group. *The Munich Project CIP (vol. 1)*, volume 183 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1985.

[Gro95] OMG (Object Management Group). The common object request broker: Architecture and specification, rev.2.0. *OMG Document ptc/96-03-04*, 1995.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1992.

[HP87] D. Hatley and I. Pirbai. *Strategies for Real-Time System Specification.* Dorset House, New York, 1987.

[Hri98] C.E. Hrischuk. *Trace-Based Load Characterization for the Automated Development of Software Performance Models.* PhD thesis, Carleton University, Ottawa, Canada, 1998.

[Hud87] S.E. Hudson. Incremental attribute evaluation: an algorithm for lazy evaluation in graphs. Technical Report 87-20, University of Arizona, 1987.

[HW95] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars. In A. Corradini and U. Montanari, editors, *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2, Amsterdam, 1995. Elsevier.

[Jac94] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, MA, fourth edition, 1994.

[KCM90] T. D. Kimura, Y. Y. Choi, and J. M. Mack. Show and tell: A visual programming language. In *[Gli90]*, pages 397–404. 1990.

[KHCM98] Stuart Kent, John Howse, Franco Civello, and Richard Mitchell. Semantics through pictures: Towards a diagrammatic semantics for oo modeling notations. In Haim Kilov and Bernhard Rumpe, editors, *Proc. ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, volume 1357 of *Lecture Notes in Computer Science*, pages 182–187, Berlin, 1998. Springer Verlag.

[KKS97] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. In *[?]*, pages 479–502, 1997.

[KS90] K. M. Kahn and V. Saraswat. Complete visualizations of concurrent programs and their executions. In *[VL90]*, pages 7–15, 1990.

[KSW95] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Sciences*, 20(1):21–51, 1995.

[Lap96] Stephan Lapalut. Expressing canonical formation rules with graph grammars operations. In *Conceptual Structures: Knowledge Representation as Interlingua, 5th International Conference on Conceptual Structures (ICCS'96), Bondi Beach, Sydney, Australia, Auxiliary Proceedings*, pages 58–69. University of New South Wales, 1996.

[McI95] David M. McIntyre. Design and implementation with vampire. In *[BGL95]*, pages 129–160. 1995.

[Mey97] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1997.

[MOTU93] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[MV95] M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *[VL95]*, pages 203–210, 1995.

[MWT94] H. A. Müller, K. Wong, and S. R. Tilley. Understanding software system using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994.

[Nag79a] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen,Implementierungen.* Vieweg, Braunschweig, 1979.

[Nag79b] Manfred Nagl. A tutorial and bibliographical survey on graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 70–126, 1979.

[Nag96] M. Nagl, editor. *Building Tightly Integrated Software Development Environments.* Number 1170 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1996.

[Ous94] John Ousterhout. *Tcl & the Tk Toolkit.* Addison-Wesley, New York, 1994. Software package TCL/TK, available from *ftp.cs.berkeley.edu:/pub/tcl.*

[Par72] David Parnas. A technique for software module specifications with examples. *Communications of the ACM*, 15:330–336, 1972.

[PBA+92] Jan Paredaens, Jan van den Bussche, Marc Andries, Marc Gemis, Marc Gyssens, Inge Thyssens, Dirk van Gucht, Vijay Sarathy, and Lawrence Saxton. An overview of good. *SIGMOD Record*, 21(1):25–31, 1992.

[Pfe95] Joseph J. Pfeiffer Jr. $ludwig_2$: Decoupling program representation from processing models. In *[VL95]*, pages 133–139, 1995.

[PR69] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *Int. Joint Confenence on Artificial Intelligence*, pages 609–619, 1969.

[Rad97] Ansgar Radermacher. Distribution of Existing Programs via CORBA. In *International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, pages 157–168, 1997.

[Rat97] Rational Software Corporation. UML semantics, version 1.1. http://www.rational.com, 1997.

[RBEL91] James Rumbaugh, Michael Blaha, William Premerlani Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rob92] Peter J. Robinson. *Hierarchical Object-Oriented Design.* Prentice Hall, Englewood Cliffs, MA, 1992.

[Roz99] Grzegorz Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications*, volume 2. World Scientific, Singapore, 199. to appear.

[Roz97] Grzegorz Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, Singapore, 1997.

[RS74] Grzegorz Rozenberg and Arto Salomaa. *L Systems.* Number 15 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1974.

[RS86] Grzegorz Rozenberg and Arto Salomaa, editors. *The Book of L*, Berlin, 1986. Springer-Verlag.

[RS97] Jan Rekers and Andy Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Ennvironments.* Springer Verlag, Berlin, 1988.

[Sch70] Hans Jürgen Schneider. Chomsky-Systeme für partielle Ordnungen. Arbeitsbericht 3,3, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen, 1970.

[Sch91] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen Anwendungsbeispiele und Werkzeugunterstützung.* Deutscher Universitäts-Verlag, Wiesbaden, 1991.

[Sch94] Andy Schürr. Rapid programming with graph rewrite rules. USENIX Symp. Very High Level Languages (VHLL),Santa Fee, New Mexico, pages 83–100, 1994.

[Sch97a] Andy Schürr. BDL - a nondeterministic data flow programming language with backtracking. In *[VL97]*, pages 394–401, 1997.

[Sch97b] Andy Schürr. Programmed graph replacement systems. In *chapter 7 in [Roz97]*, pages 479–546. 1997.

[Sch98] A. Schürr. *The PROGRES Language Manual Version 9.x.* RWTH Aachen, Ahornstr. 55, D-52074 Aachen, Germany, 1998. See: http://www-i3.informatik.rwth-aachen.de/ research/progres/ProgresSyntax.html.

[SCS94] David Canfield Smith, Allen Cypher, and Jim Spohrer. Kidsim: Programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, 1994.

[Ste94] Nancy B. Stern. *Structured COBOL programming.* Wiley, 1994.

[SW97] A. Schürr and A.J. Winter. Formal definition and refinement of uml's module/package concept. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology — ECOOP '97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 211–215, Berlin, 1997. Springer Verlag.

[SWZ95a] Andy Schürr, Andreas Winter, and Albert Zündorf. Visual programming with graph rewriting systems. In *[VL95]*, pages 195–202, 1995.

[SWZ95b] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In Wilhelm Schäfer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC'95)*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234, Berlin, 1995. Springer Verlag.

[TB94] Gabriele Taentzer and Martin Beyer. Amalgamated graph transformations and their use for specifying AGG — an algebraic graph grammar system. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 380–394, 1994.

[Tre96] M. Tresch. Middleware: Key technology for the development of distributed information systems. *Informatik-Spektrum*, 19(5):249–256, 1996. (in german).

[Tyu91] Enn Tyugu. Higher order dataflow schemas. *Theoretical Computer Science*, 90:185–198, 1991.

[VL90] *Proc. IEEE Workshop on Visual Languages (VL'90)*, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[VL93] *Proc. IEEE Symposium on Visual Languages (VL'93)*, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[VL95] *Proc. IEEE Symposium on Visual Languages (VL'95)*, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[VL97] *Proc. IEEE Symposium on Visual Languages (VL'97)*, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[VW86] G. M. Vose and G. Williams. LabVIEW: Laboratory virtual instrument engineering workbench. *BYTE*, 11(9):84–92, 1986.

[WC96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Mateo, CA, 1996.

[Wes96] Bernhard Westfechtel. A graph-based system for managing configurations of engineering design documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4):549–583, 1996.

[WS84] Richard M. Wiener and Richard Sincovec. *Software Engineering with Modula-2 and Ada*. John Wiley, New York, 1984.

[WS97] A.J. Winter and A. Schürr. Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems. Technical Report AIB 97-3, Dept. of Computer Science, RWTH Aachen, 1997.

[You89] E. Yourdon. *Modern Structured Analysis*. Yourdon Press, Englewood Cliffs, 1989.

[Zam96] A. Zamperoni. Grids - graph-based, integrated development of software: Integrating different perspectives of software engineering. In *Proc. ICSE '18, Int. Conf. on Software Engineering*, pages 48–59, Los Alamitos (CA), 1996. IEEE Computer Society Press.

[Zün96] A. Zündorf. *Eine Entwicklungsumgebung für programmierte Graphersetzungssysteme*. Deutscher Universitätsverlag, Wiesbaden, 1996. Dissertation, RWTH Aachen.

[ZZ97] D.-Q. Zhang and K. Zhang. Reserved graph grammar: A specification tool for diagrammatic vpls. In *[VL97]*, pages 288–295, 1997.