PROGRES for Beginners

Andy Schürr

Lehrstuhl für Informatik III
RWTH Aachen, Ahornstr. 55
D-52074 Aachen, Germany
email: andy@i3.informatik.rwth-aachen.de

1. Introduction

Graphs play an important role within many areas of applied computer science, and there exists an abundance of visual languages and environments which have graphs as their underlying data model [41, 42, 43]. Furthermore, *rule-based languages* and systems haven proven to be well-suited for the description of complex transformation or inference processes on complex data structures. And *grammars*, for instance in the form of attribute grammars or definite clause grammars, are often used to describe syntax and semantics of textual languages and to generate parsers for them.

Although graphs and rule-based systems or grammars are quite popular among computer scientists, their combination in the form of graph rewriting systems or graph grammars is more or less unknown. At least one reason for this short-fall is that *graph grammar research* was from its very beginning in the early 70's focused on producing theoretical results, and working implementations based on these concepts were not available for a very long time. Furthermore, many people believe that modeling with graphs and graph rewriting systems leads to inherently inefficient implementations due to the NP-completeness of many graph algorithms. This situation changed gradually with the appearance of first graph rewriting system or graph grammar *implementations* like GraphEd [19], PAGG [18], and GOOD [17].

The essential idea of all implemented graph grammar or graph rewriting system approaches is to be a generalization of string grammars or term rewriting systems. The terms "graph grammar" and "graph rewriting system" are often used as synonyms to each other. But strictly speaking, a *graph grammar* is a set of productions that generates a language of terminal graphs and produces nonterminal graphs as intermediate results. And a *graph rewriting system* is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs without distinguishing terminal and nonterminal results. Graph grammars are mainly used for synthesizing or recognizing graph-like data structures in biology, chemistry, and other research areas. Graph rewriting systems, on the other hand, are often used as *visual and executable specifications* of abstract data types or graph manipulating tools (cf. [4, 8, 9, 11]).

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

- for specifying tools and data structures of integrated software engineering environments [10],
- for describing process modeling, version control, and configuration management tools in CIM environments [40, 20],
- as the underlying fundament of a new approach to diagram parsing [23],
- and finally for defining the semantics of a visual database query language [2].

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

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

This paper is an *informal introduction* to currently distributed version of *PROGRES*. We will use a single running example, the specification of a small software design and configuration tool, to present its most important constructs and properties. Although being a kind of "toy" example compared with real software engineering tools, it is complex enough to discuss all important features of the language PROGRES. A complete formal description of PROGRES and its underlying graph rewriting formalism is outside the scope of this paper and may be found in [36, 37].

The paper is *organized* as follows: section 2 gives a rather short overview of other tool specification languages, discusses their deficiencies and motivates thereby why it was necessary to develop a specification language like PROGRES. Section 3 explains our running example. Section 4 shows how software engineering documents are internally represented by attributed graphs and how graph schema definitions may be used to characterize the static properties of a class of internal documents. Section 5 and section 5 deal with the definition of rather complex document querying and manipulating processes as subgraph searching and rewriting programs, whereas section 7 discusses advanced features for improving the runtime behavior of specifications and for writing highly reusable specifications. Section 8 summarizes finally the main characteristics of our specification language.

2. Other Tool Specification Languages

The specification language PROGRES is a strongly and even almost statically typed language that is based on the concept of *programmed graph rewriting systems*. Considering its expressiveness and its derivation history, the underlying formalism belongs to the *algorithmic graph grammar approach* [31]. To our knowledge, it is the first approach to evolve the theoretically well-founded calculus of programmed graph rewriting systems into a technique that is applicable to the specification of complex software engineering tools.

Other *graph grammar based specification languages* proposed by Göttler [16, 16], Kaplan and Goering [25] or Schütte [39] are more limited in their expressiveness and do not possess any kind of typing concept. Furthermore, they do not support the definition of derived relationships and have no or much more primitive control structures. Finally, all of them attach the definition of derived attributes to graph rewrite rules instead of keeping them separate as PROGRES does, i.e. they relate attribute derivations to the derivation of parse trees. They are therefore more suited for syntax analysis of graph classes than for specifying complex document manipulating software engineering tools.

Comparing our approach with the rather popular formalism of *attribute tree grammars* used in other software engineering environment projects (e.g. in the Cornell Synthesizer Generator [34] and in PSG [35]) or in compiler generator projects (e.g. in OPTRAN [28]), we identify at least three principal differences:

- Over here, *directed attribute equations* are used to describe attribute dependencies between adjacent graph nodes. They do not describe attribute flows along the edges of a parse tree, as is customary in attribute tree grammars [24, 34]. Productions for generating parse trees only are not sufficient for specifying tool operations which manipulate or analyze already existing software document parts. We, therefore, believe that the specification of graph schemata and *attribute dependencies* and that of *graph rewriting rules* should be *kept separate* from each other.
- Of course, in an attributed graph it is impossible to identify directions like "up" and "down". Consequently, our derived attributes are *not classified into inherited and synthesized attributes* as they are in attribut tree grammar projects [34, 35]. Thus the development of nonnaive (incremental) *attribute evaluation algorithms* is much more difficult than in the case of attribute tree grammars.
- Our *graph model*, which is *more general* than attributed trees, frees the specificator from the somewhat artificially different treatment of relationships representing either context-free or context-sensitive syntax, the latter one also called static semantics. Using (derived) labeled directed edges in particular does not enforce putting context-sensitive information into very complex structured attributes, which is a severe handicap for any incremental attribute evaluation algorithm [32].

Some research has been undertaken to *overcome the disadvantages* and principal restrictions of the "classical" *attribute tree grammar formalism* in modeling context-sensitive relations [21]. This research still adheres to the data model of an attributed tree, proposes the introduction of remote references (relations) and/or tree-valued attributes, and still suffers from the above-mentioned problems, when specifying inherently graph-like structures and operations (cf. example of section 3).

Comparing the approach presented here to the research on *conceptual modeling in the database area*, PROGRES specifications can be viewed as conceptual database specifications, which determine the static structure as well as the allowed dynamic behavior of a database. Our underlying data model has a lot of similarities to so-called semantic data models [22], and especially to the binary relationship model [14], which is a variant of the well-known Entity Relationship model. However, while in the database area, the specification is usually restricted to modeling the static structure of a database or to define queries for an already existing database [1, 2], our specification language also offers sophisticated features for specifying the allowed *dynamic behavior of a database* in a rule-oriented graphical style. In this sense, PROGRES may be considered a visual database programming language, which is similar to the graph grammar based database manipulation language GOOD [17]. It offers additional means for defining derived data and for nondeterministic programming.

Taking into account that PROGRES is a graphical and executable specification language, it might be useful to have a final look onto *visual rule-oriented languages* like BITPICT [15] and ChemTrains [3]. Their deficiencies come from the following sources: They have their main focus on *manipulation of data structures* and data definition sublanguages are not provided. Even a language like VAMPIRE with its class hierarchies and icon rewriting rules [29] comes without a rigid type concept and *without any type checking tools*. Therefore, all these languages postpone recognition of programming errors to runtime and suffer from the same disadvantages as any untyped or weakly typed programming language.

To summarize, the overall argument for using PROGRES in IPSEN-like projects is that *graph structures appear frequently* within SDEs and other systems which handle complex data structures. These graph structures require a suitable specification mechanism covering *static as well as dynamic aspects of complex data structures to the same extent*. The specificator should not be forced to mentally transform a real-world graph-like model into a tree-like one in order to do the specification. Furthermore, he should not be forced to write "code" which checks and preserves static properties of these data structures instead of specifying them in a completely declarative manner. But the main argument for PROGRES is, in fact, its *generality*. It can be applied in any situation where complex graph-like structures have to be modeled and queries as well as update operations on these structures have to be specified.

3. The Running Example

As an example for motivating and explaining our graph technology approach, we assume the development of a particular software engineering tool. It is a syntax-directed *software design editor* with included configuration managment functionality (cf. [27]). In order to keep the resulting specification as simple as possible, its supported *module interconnection (MIL) language* is a subset of the language which is used within the IPSEN project. It offers three types of modules, namely data type modules, data object modules, and function modules, and distinguishes between interface imports and realization (body) imports. It has no support for building subsystems or for using concepts like inheritance or genericity which are a strict necessity in practice.

Fig. 3.1 displays a *single software architecture* which contains examples for all abovementioned concepts. A functional `Main` module has one realization variant which needs an abstract data object module `UserInterface` and an abstract data type module `Files`. The implementation of the module `Files` is independent from any available window system platform. It has one variant for `UNIX` like operating systems and one variant for `MSDOS` systems. The other module `UserInterface` imports the module `Files` in its interface (its interface operations have parameters of type `File`). Its only available implementation variant uses a `UNIX` implementation of the window system `X`. As a consequence, only one consistent configuration of the whole system exists. It consists of variant `V1`of `Main` (which has no restrictions concerning an underlying operating system or window system), the variant `V1` of the module `UserInterface`, and the variant `V1` of themodule `Files`.

The extension of the chosen module interconnection language for *variants of module realizations* with different body imports and properties is *motivated* as follows: in general, the configuration of a consistent system implementation with a required set of properties is a

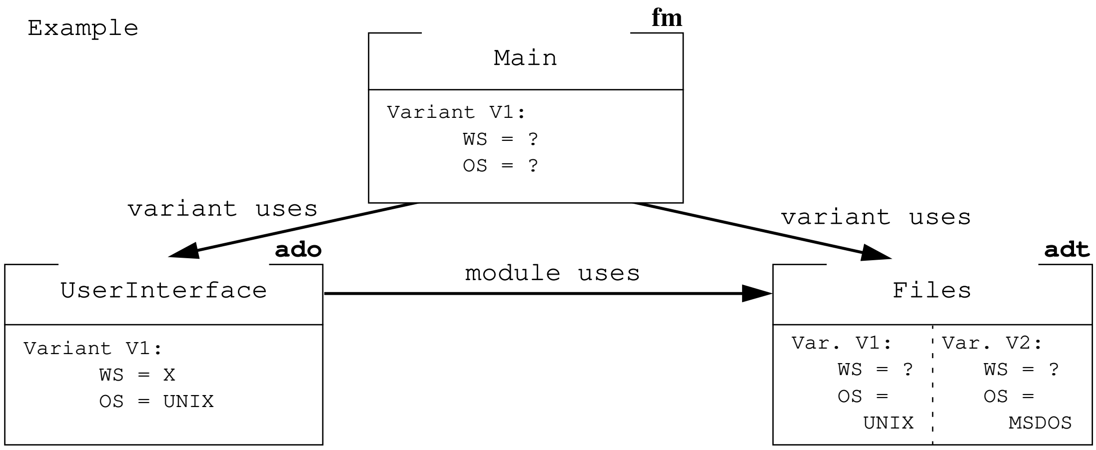


Fig. 3.1. A sample architecture with three modules and four variants

rather difficult task. It shows very well that graph rewrite rules with complex application conditions and control structures with backtracking are rather useful for specifying software development tasks. It has the consequence that the software design document `Example` of fig. 3.1 models a *family of system designs* instead of representing a single design decision, where all possibly existing module variants share their implementation imports.

Finally, please note that our design editor maintains information about “sizes” of modules, variants, systems, etc. These *sizes are derived data* which will be computed from text (file) lengths of module interface descriptions as well as implementation variants. They are a good example for explaining the PROGRES language’s support for derived attributes. The following basic text sizes are assumed in the sequel (in lines of code):

`Main` (interface spec.):	100	variant `V1`:	2000		
`UserInterface`:	1000	variant `V1`:	5000		
`Files`:	500	variant `V1`:	3000	variant `V2`:	2000

4. Definition of Graph Schemata

Generally speaking, the internal representation of all software documents is a *directed attributed graph*. Such a graph consists of labeled nodes and directed labeled edges, where attributes may be attached to nodes only. Node and edge labels distinguish different types of objects and relationships between objects, respectively. They are called *node types and edge types* in the sequel. *Attributes* are needed to store additional information that is not necessarily to be represented in the graph structure as, for example, the detailed structure of module names. In other words, attributes represent the information that is local to a particular node and which has an unimportant structure from the current point of view. In our simple running example even all properties of a module variant or a system configuration are stored in a single set-valued attribute.

Fig. 4 shows the *internal graph representation* of the software system in fig. 3.1, in the sequel called MIL graph. Any module and any module variant is modeled as an own node; `has`-edges connect module nodes to their implementation variants. The whole software system is represented by a single node of type `System` with `contains` edges to all existing module nodes. An additional `Config` node stores all available information about the only currently existing consistent system configuration. It has the `Name "C"`, needs a `UNIX` implementation of the window system `X`, and has a `Size` of 10000 lines of code. It `contains` one variant of each module.

The language features provided by PROGRES to define static properties of MIL graphs in the form of a graph schema will now be explained in detail. Note that we use the term *graph schema* in the same sense as the term database schema is used in the database design process.

PROGRES offers the following syntactic constructs for defining the components of a particular class of graphs and their legal combinations. These are

- *node types* like `System` or `FunctionModule`, which determine the static properties of their nodes instances,
- *intrinsic relationships*, henceforth called *edge types* like `v(ariant body)_uses` or `m(odule interface)_uses`, which are explicitely manipulated and possess restrictions concerning the types of their sources and targets,
- *derived relationships*, which model often needed paths of a given graph, as for instance the path from a particular module to all directly or indirectly needed other module implementations, and which are defined by means of *path expressions*,

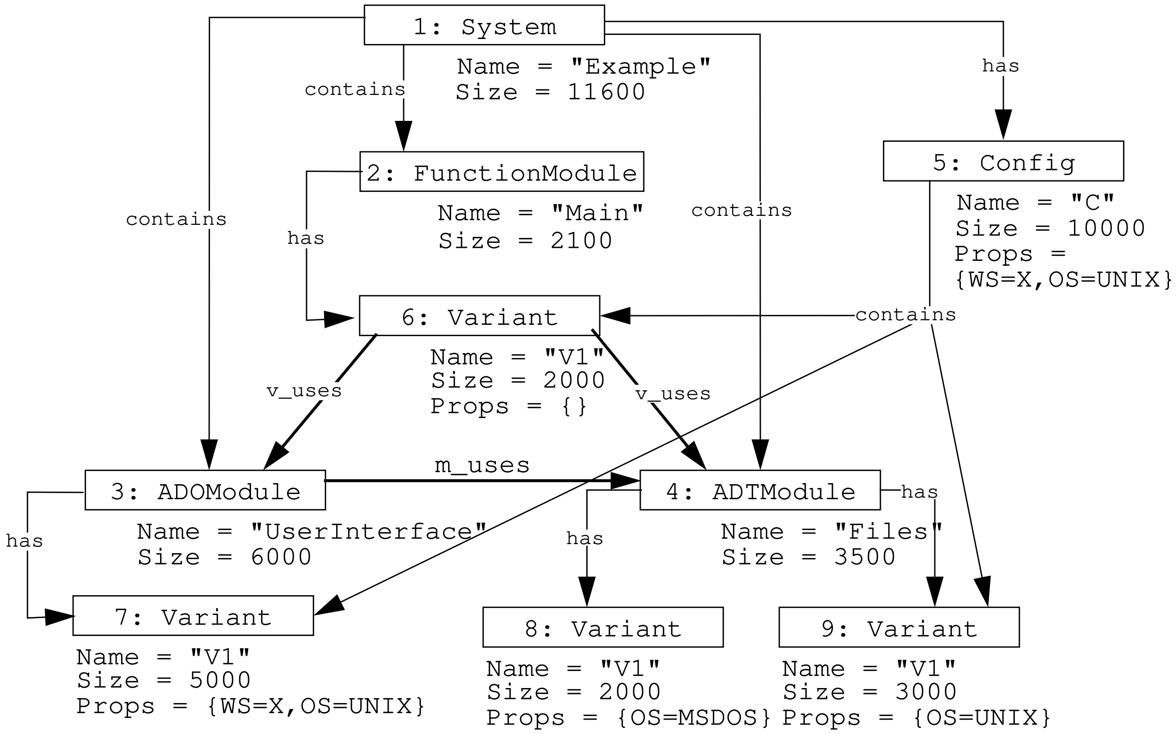


Fig. 4: Internal "MIL" graph representation of sample architecture

- *intrinsic attributes* like the `Name` of a system component, which are defined for a particular set of node types and which are explicitely manipulated, too,
- and finally *derived attributes* like the `Size` of a system component, which are defined by means of directed equations and which may have different definitions for different 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 node types `FunctionModule`, `ADTModule`, and `ADOModule`. Nodes of all these types are legal sources of `has` and `m_uses` edges, legal targets of `contains`, `m_uses`, and `v_uses` edges, and they possess `Name` and `Size` attributes. As we will see later on, even the definition of the `Size` attribute is the same for all three types of nodes. Therefore, it was natural to introduce an additional concept into PROGRES, 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*, which are from a theoretical point of view types of node types. Compared with abstract syntax definition notations, node classes play about the same role as phyla or nonterminal classes, whereas node types are the counterparts to operators or terminal classes (cf. [34]).

Rather than just introducing node classes as second-order types, PROGRES enables us to build *hierarchies of node classes* by exploiting *multiple inheritance* as the relation between node classes. As usual in object-oriented languages, PROGRES uses the *is_a* notion to express the inheritance relation. *Multiple inheritance* may be used to cut down the size of graph schema definitions considerably. This fact is shown in fig. 5, which contains a *graphical representation* of our MIL graph schema. It has to be read as follows:

- *Normal boxes* represent node classes which are connected to their superclasses by means of *dotted edges* representing "is-a" relationships; `ATOM` is for instance a subclass of `UNIT`.

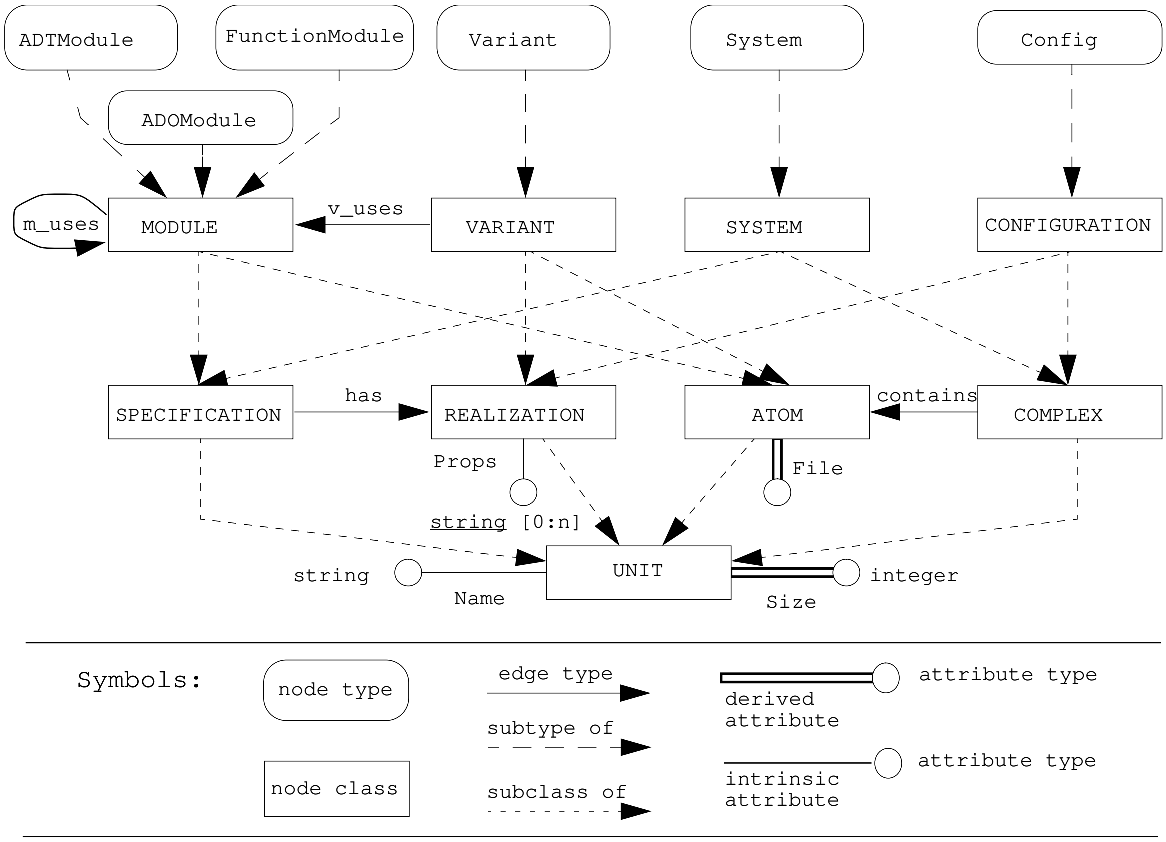


Fig. 5: The graph schema of MIL graphs (without derived relationships)

- *Boxes* with *round corners* represent node types which are connected to their uniquely defined classes by means of *dashed edges* representing "type is instance of class" relationships; the type `ADTModule` belongs for instance to the class `MODULE`.
- *Solid edges* between node classes represent edge type definitions; the edge type `v_uses` is for instance a relationship between `VARIANT` nodes and `MODULE` nodes and `m_uses` edges connect `MODULE` nodes with other `MODULE` nodes.
- *Circles* attached to node classes represent attributes with their names above or below the connection line segment and their type definiton nearby the circle. A double line segment connects a derived attribute like `Size` to its node class and a normal line segment connects an intrinsic attribute like `Name` to its class.

The root class of our inheritance hierarchy is positioned at the bottom of fig. 5. It has the name `UNIT`. It possesses an intrinsic `string`-valued attribute `Name` and a derived `integer`-attribute `Size`. *All nodes* within MIL graphs are *instances* of node types which belong indirectly to the class `UNIT`. As a consequence, they are owners of `Name` and `Size` attributes. But we will see later on that different node types have different rules how to compute their `Size`.

The rest of the graph schema is based on the *distinction between* `SPECIFICATION` and `REALIZATION` *nodes* as well as on the distinction between `COMPLEX` and `ATOM(ic)` nodes. Any `SPECIFICATION` `has` an arbitrary number of `REALIZATION` nodes. `REALIZATION` nodes, on the other hand, are characterized by certain implementation properties, which are stored as a set of strings in the attribute `Props`. Any `COMPLEX` node `contains` a set of `ATOM` nodes which have in turn a `File` attribute to refer to their contents. It is worthwhile to notice that `has` edges and `contains` edges form flat hierarchies which share their leaf nodes. Therefore, it would be rather difficult to model such a software design document as an attributed tree, even if import/export relationships are disregarded (see below).

All remaining node *classes* are then just *combinations* of these four classes and inherit their properties. A `SYSTEM` is a complex specification of a software system architecture. A `CONFIGURATION` is a realization of a complex software system with certain properties. A `MODULE` is an atomic specification of a system component and a `VARIANT` is an atomic module realization with certain properties. `MODULE` and `VARIANT` nodes have additional `m_uses` and `v_uses` edge type definitions. This expresses the fact that a module (interface) specification may import types from an arbitrary number of different modules and that realization variants may import additionally needed resources from other modules.

The top layer of fig. 5 introduces finally all *needed node types*. `MODULE` is the only node class with more than one node type, since we did not distinguish variants of different module types and since there is only one type of system or configuration, respectively.

If we add an additional smallest class, the empty set of node types to our class hierarchy then any two classes have to possess a uniquely defined least upper common superclass and a uniquely defined least lower common subclass. This requirement that PROGRES *class hierarchies* have to be *lattices* facilitates the detection of misuses of the multiple inheritance concept and simplifies the task of type checking PROGRES specifications considerably.

It is often the case that unions or intersections of node sets are needed, where one node set contains instances of a node class `C1` and the other node set contains instances of a node class `C2`. Assuming lattices as inheritance hierarchies, we can use the least upper common superclass of `C1` and `C2` as the type of the set union and the least lower common subclass as the type of the set intersection. Let us assume that we have a set `S1` of `SPECIFICATION` nodes and another set `S2` of `ATOM` nodes as well as

`S := S1 or S2` and `S' := S2 and S2`.

In this case the derived type information for `S` is `UNIT` and for S' is `MODULE` (the operator symbol `or` denotes union and the operator symbol `and` denotes intersection).

To complete the explanation of language features for defining graph schemata, we have to describe how edge cardinalities, attribute types, and attribute evaluation rules may be defined. For this purpose we switch from the graphical schema definition notation of fig. 5 to the more detailed *textual representation* of fig. 7.

Standard attribute domains like `integer`, `string` , and `boolean` together with their functions are a built-in part of the language PROGRES. All other types, as for instance the type `file` of the `ATOMIC` node attribute `File`, must be defined elsewhere in a so-called “host” language (Modula-2 or C). Special *import clauses* signal their existence and provide necessary type checking informations (cf. fig. 6).

Additional *functions* over imported or built-in types and functions may be defined in an *application-oriented style* of programming. Fig. 6 contains the definition of three functions (`select`, `merge`, and `addSize`). The first two deal with configuration or module variant properties, which are represented as (sets of) strings of the form `"name:value"`, in short called property pairs. These functions use an elsewhere defined function `propName`, which selects the substring of its argument preceding `":"`. The function `select` gets a particular property name of type `string` and a set of property pairs as input and returns all those pairs which have the given name as first component.

The *cardinality clause* `[0:n]` within the declaration of `PSet` is a hint for the type checker that its actual value may be an arbitrary set of strings, including the case of an empty set. A nonempty set of strings would have the type definition `string [1:n]` , and an either

```
from Files import

   types  file;

   functions
      size: ( file ) -> integer;

end;

function select: ( PName: string ;
                   PSet: string [0:n] ) -> string [0:n] =
   use P: string := elem( PSet ) ::
       [ propName( P ) = PName :: P | nil ]
   end
end;  (* Returns all ('PName', ?) pairs in a given property set *)
      (* 'PSet' which have 'PName' as their first component.    *)

function merge: ( PSet1, PSet2 : string [0:n] ) -> string [0:n] =
   use NSet1: string [0:n] := propName( PSet1 ) but not propName( PSet2 );
       NSet2: string [0:n] := propName( PSet2 ) but not propName( PSet1 ) ::
       select( NSet1, PSet1 ) or select( NSet2, PSet2 )
                              or ( PSet1 and PSet2 )
   end
end;   (* The result requires any property which is         *)
       (* 1) required in 'PSet1' and unspecified in 'PSet2' *)
       (* 2) required in 'PSet2' and unspecified in 'PSet1' *)
       (* 3) required in 'PSet1' and 'PSet2'.                *)

function addSize: ( Val : integer ; Atom : ATOM ) -> integer =
   Val + Atom.Size
end;
```

Fig. 6: Import of external types and functions as well as definition of local functic

undefined or uniquely defined string parameter the type definition `string [0:1]`. Remains the case [1:1], the default, for an always defined and single valued parameter (or local variable). In a similar way, `-> string [0:n]` indicates that our function returns a possibly empty set of values and not just a single `string` value.

The body of the function `select` defines a *local variable* `P`, which gets any possible element of `PSet` as its value. More precisely, one element of `PSet` after the other is assigned to `P` and the line

```
[ propName( P ) = PName :: P | nil ]
```

has to be read as "include `P` into the result set if its name component is equal to `PName`, otherwise include nothing into the result set".

Please note that it is not required, or better even not possible, to program an explicit loop over the elements in `PSet` within functions. It is an inherent property of the language PROGRES that it deals properly with any kind of *nondeterminism*, in this case with the nondeterministic selection of a set element and its assignment to the single value containing variable `P`. From a specifier's point of view it makes no difference whether a call to function `select` computes the return values for all possible assigments to the variable `P` immediately (the calling function selects then needed elements from the result set) or whether it returns first an arbitrarily chosen value and computes additionally needed values later on (if the calling function needs all possible result values or a result value with a certain property) as long as the set of results is finite.

The following function `merge` makes extensive use of the PROGRES *set operators* `and` (intersection), `or` (union), and `but not` (difference). Its first line

```
NSet_1 := propName(PSet_1) but not propName(PSet_2)
```

computes the set of all property names used within `PSet_1` not used in `PSet_2` and assigns it to the variable `NSet_1`. The next line does the same with exchanged indices `1` and `2`. Afterwards, the union of property pair sets is built which are either exclusively defined in the first set or the second set or which have consistent definitions in both sets.

Any call to the function `select` with a set of property names as its first actual parameter is *implicitly expanded to the union* of all possible function calls with different set elements as a first parameter (since its first formal parameter is not declared as a set). A call of the form

```
select({"WS", "OS"}, PSet)
```

is for instance equivalent to

```
select("WS", PSet) or select("OS", PSet).
```

The function `merge` will be used later on to combine a required set of properties of a partially constructed system configuration with the requirements of a still needed module variant. Let us assume that we start with the requirement of the window system `"X"` and select a module variant which makes no assumptions about an underlying window system, but requires either a `"UNIX"` or `"MSDOS"` operating system. This leads to the call of

```
merge({"WS:X"}, {"OS:UNIX", "OS:MSDOS"})
```

with the result (cf. case 1 and case 3 of `merge` comments in fig. 6)

```
{"WS:X", "OS:UNIX", "OS:MSDOS"}.
```

Let us assume furtheron that the next selected module variant requires the operating system "UNIX". This leads to the call of

```
merge({"WS:X", "OS:UNIX", "OS:MSDOS"}, {"OS:UNIX"})
```

which returns the result (cf. case 1 and 2 of `merge` comments in fig. 6)

```
{"WS:X", "OS:UNIX"}.
```

The last declaration of fig. 6 defines an auxilary function `addSize`, which takes a node of class `ATOM` as input and adds its `Size` attribute value to another given `integer` value. This function is later used in fig. 7 to compute derived `Size` attributes of `COMPLEX` nodes.

Fig. 7 contains a cut-out of the *textual graph schema representation* which contains more information than the graphical schema declaration of fig. 5. It reveals additional cardinality constraints for edges like “any `SPECIFICATION has` an arbitrary number of `REALIZATION` nodes, but a `REALIZATION` belongs always to a single `SPECIFICATION`”. More precisely, the same four *cardinality constraints* are available for the definition of edge types as presented above for the definition of formal function parameters. A cardinality constraint of the form `[min:max]` behind the target class within an edge type declaration defines lower and upper boundaries for the bundle of outgoing edges at a single source node. Such a constraint behind the source class defines lower and upper boundaries for the bundle of incoming edges at a single target node.

The most important part of fig. 7 are the derived *attribute evaluation rules*. The default rule within the declaration of the `Size` attribute requires that “pure” `UNIT` nodes have no size at all. It is only a part of the specification for completeness reasons and could be omitted. All node types of our example, which have the `Size` attribute, inherit a *redefined* evaluation rule. For demonstration purposes, we assume that `SPECIFICATION` nodes have a size which is the maximum of the sizes of their realizations. Furthermore, `COMPLEX` nodes have a size which is the sum of all their component sizes, and `ATOMIC` nodes compute their size from the contents of their `File` attribute.

```
node class UNIT
   derived
      Size : integer = 0;
   intrinsic
      Name : string := "";
end;     (* Root of class hierarchy. *)

node class SPECIFICATION is a UNIT
   redef derived
      Size = max( 0, all self.-has->.Size );
        (* The 'Size' of a specification is the maximum of    *)
        (* the size of all its realizations, instead of       *)
        (* being the sum (which would reasonable, too).       *)
end;    (* A 'SPECIFICATION' is either the complete design of *)
        (* a software system or a design of one its parts.    *)

node class REALIZATION is a UNIT
   intrinsic
      Props : string [0:n];  (* Set of guaranteed properties *)
end;    (* A 'REALIZATION' is an implementation of a 'SPEC.' *)
        (* which fulfills a given set of properties like     *)
        (* needed hardware platforms, operating system, ...  *)

edge type has : SPECIFICATION [1:1] -> REALIZATION [0:n];
        (* A 'SPECIFICATION' 'has' an arbitrary number of    *)
        (* of 'REALIZATIONS', but a 'REAL.' belongs to one   *)
        (* uniquely defined 'SPEC.'.                         *)

node class COMPLEX is a UNIT
   redef derived
      Size = addSize( 0, all self.-contains-> );
       (* The 'Size' of a complex unit is the sum of *)
       (* the 'Size' of all its children.            *)
end;

node class ATOM is a UNIT
   intrinsic
      File : file;    (* Pointer to externally stored file. *)
   redef derived
      Size = size( self.File );
              (* 'Size' is the length of the attached File. *)
end;

edge type contains : COMPLEX [1:1] -> ATOM [0:n];

node class SYSTEM is a SPECIFICATION, COMPLEX
   redef derived
      Size = addSize( 0, all self.-contains-> );
           (* Resolves inheritance conflict of attribute   *)
           (* definitions in 'SPECIFICATION' and 'COMPLEX' *)
           (* by prefering definition in 'SPECIFICATION'.  *)
end;

node class CONFIGURATION is a REALIZATION, COMPLEX end;
     (* A 'CONFIGURATION' is a set of variants of module    *)
     (* realizations which fulfill all required properties. *)

node class MODULE is a SPECIFICATION, ATOM
  redef derived
     Size = size( self.File ) + max( 0, all self.-has->.Size );
          (* Resolves inheritance conflict of attribute    *)
          (* definitions in 'SPECIFICATION' and 'COMPLEX'  *)
          (* by building the sum of both definitions which *)
          (* are in conflict to each other.                *)
end;      (* A 'SYSTEM' contains a set of 'MODULES' which  *)
          (* have 'VARIANTS' as their realizations.        *)

node class VARIANT is a REALIZATION, ATOM end;
```

Fig. 7: Defining derived node attributes

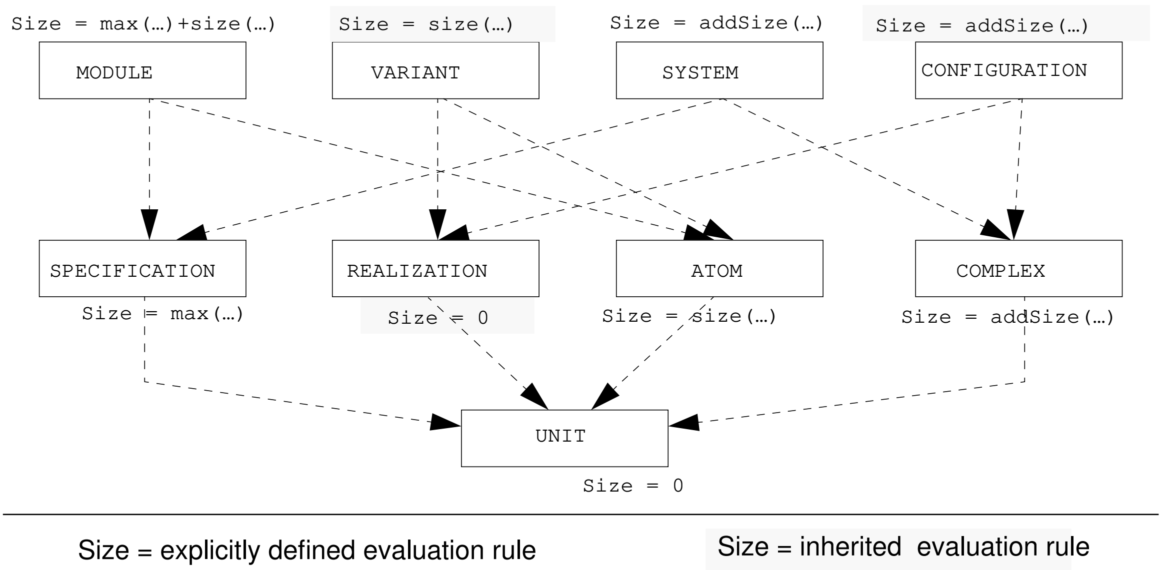


Fig. 8: Various forms of inheritance conflicts and conflict resolution strategies

As a consequence, SYSTEM nodes as well as MODULE nodes inherit two different evaluation rules from their superclasses (see also fig. 8). This constitutes a so-called *inheritance conflict* which has to be resolved manually. In the case of the class SYSTEM it seems to be logical to use the same evaluation function as in the class COMPLEX, i.e. one of the inherited evaluation rules *overwrites* the other one. In the case of the class MODULE the new rule has to be a *combination* of the rules which constitute the inheritance conflict. It computes just the sum of the two inherited evaluation rules.

The two remaining classes VARIANT and CONFIGURATION exemplify a slightly different situation. They inherit one evaluation rule from the class UNIT directly and another rule from a subclass of UNIT (VARIANT inherits a redefinition from ATOM and CONFIGURATION a redefinition from COMPLEX). In this case, PROGRES and many object-oriented programming languages *resolve the conflict automatically* in favor of the more specific rule (method).

The values of derived attributes are automatically kept in a consistent state. For this purpose an *incrementally working lazy attribute evaluation mechanism* is used. It is a variant of the well-known two-phase attribute evaluation algorithm in [23], which is an integral part of our graph-oriented database system GRAS [26]. It works as follows applied to the sample graph of fig. 4. At the beginning all derived attributes have uncomputed values. A read access to the Size attribute of the System node triggers then the evaluation of all derived attributes, which are needed for the computation of the directly needed attribute. In our case, all Size attributes (except the attribute of the Config node 5) are needed and evaluated in the correct order of their dependencies. Variant nodes 6 through 9 compute the sizes of their files. Afterwards, Module nodes 2, 3, and 4 compute the sum of their own file size and the maximum of the sizes of their variants. The System node itself computes finally the sum of the sizes of all its modules.

Afterwards, any *modification* of the given MIL graph *invalidates* all those computed *attributes* whose values *maybe affected* by the changes. Adding for instance a new variant to the module Main would affect the attribute values of the module Main itself and the System node. For further information concerning the evaluation of derived attributes see [26].

In summary, so far we have explained the PROGRES language features to define graph schemata with node types, edge types, and attributes as basic components and with node classes as a means for grouping node types with common properties. We shall see within the next two sections 5 and 6 how graph schema definitions may be checked against graph queries and rewrite rules, thereby guaranteeing that any attempt to construct a *schema inconsistent graph* is detected at *compile time*. Furthermore, we shall see in sections 6 and 7how node classes are used to define *generic graph rewrite rules* that have actual node type parameters and formal node class parameter types. Finally, we should mention that we have postponed the definition of derived relationships to the following section, where they are actually needed within graph queries.

5. Definition of Graph Queries

The graph schema definition part of a PROGRES specification as introduced in the last section enables us to specify static properties of any class of directed, attributed, node and edge labeled graphs. Using these graphs as the internal representations of documents implies that all document processing tool operations such as retrieving any kind of information from a document or modifying any part of a document can be described by *subgraph selection and subgraph replacement steps*. Analogously to the use of the term schema, in the sense of a database schema, these operations are considered to be *transactions* in the sense of database update operations. They fulfill the usual requirements for atomicity and consistency preservation as well as the additional requirements for isolation and duration in the case of multiuser access to a graph database (see discussion below).

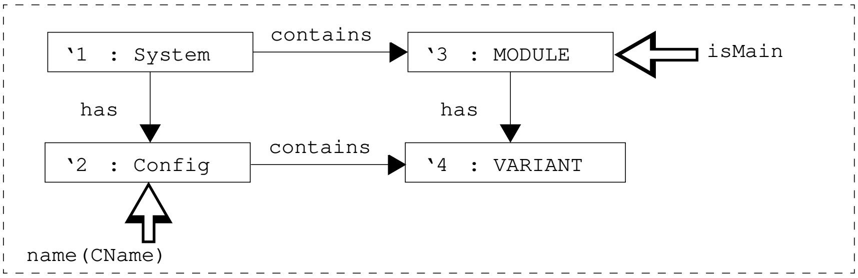
Such a document processing operation or transaction is usually composed of basic subgraph matching and replacement steps, which are specified by means of so-called subgraph *tests* and subgraph rewriting *productions* (rewrite rules). In this section, we start with the explanation how tests and productions are specified themselves and introduce then further concepts to compose these basic operations to more complex graph queries or graph modifying transactions.

Fig. 9 contains representative examples for those PROGRES language constructs which may be used to formulate *complex queries*. The most important components are *subgraph tests* which search for the existence or nonexistence of a certain subgraph pattern in a host graph. Please note that a selected subgraph in the host graph contains a corresponding node and edge for any (positive) node and edge of the regarded pattern. It may contain *additional edges* between selected nodes in the host graph. All subgraph tests use so-called *restrictions* (double arrows pointing to a node) and *paths* (double arrows between two nodes) as complex application conditions. They are defined later on in fig. 11.

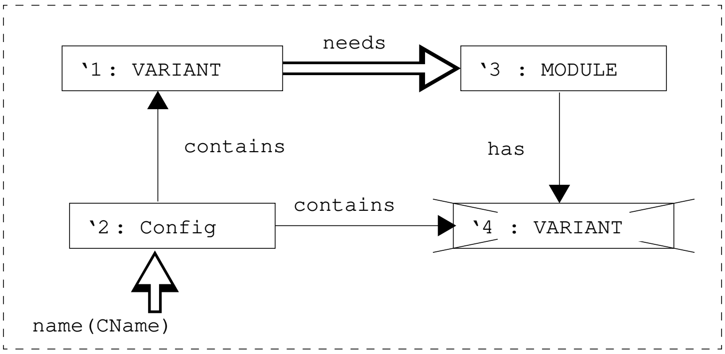
The test `ConfigurationWithMain` matches any subgraph consisting of the following nodes and edges by binding its identifiers `'1`, `'2`, etc. to nodes of the host graph:

- the identifier `'1` is bound to a node of the type `System`,
- the identifier `'2` is bound to a node of type `Config` which is the target of a `has`-edge from the `System` node,
- the identifier `'3` is bound to a node of (a type belonging to) class `MODULE`, which is part of the selected `System` and fulfills the additional *restriction* `isMain` (expressed by the double arrow pointing to node `'3`; cf. fig. 11),
- and the identifier `'4` to a node of (a type of) the class `VARIANT`, which is part of the selected configuration and a realization variant of the selected main module.

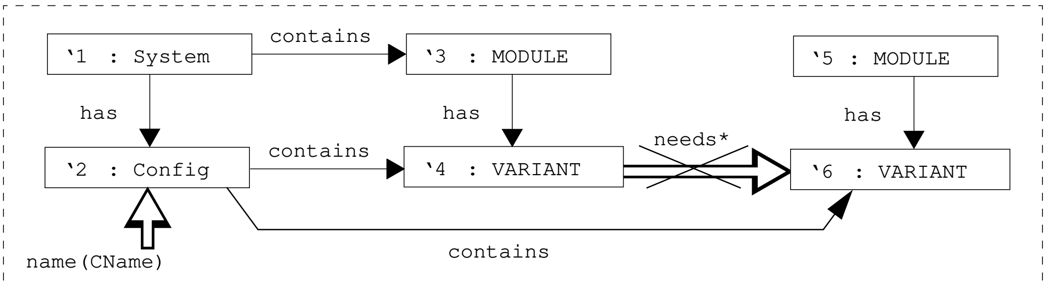
```
query ConsistentConfiguration( out CName : string ) =

   (* A configuration is consistent if:
   (* 1) it contains a variant of the system’s main module, *)
   (* 2) it contains a variant for any module which is      *)
   (*    needed by another included variant, and            *)
   (* 3) it does not contain variants which are not needed  *)
   (*    by needed variants.                                *)

   use LocalName: string do
      ConfigurationWithMain( out LocalName )
    & not UnresolvedImportExists( LocalName )
    & not ConfigurationWithUselessVariant( LocalName )
    & CName := LocalName
   end

end;

test ConfigurationWithMain( out CName : string )  =
```




```
end;

test UnresolvedImportExists( CName : string ) =
```




```
end;

test ConfigurationWithUselessVariant( CName : string )  =
```




```
end;
```

Fig. 9: Specification of atomic subgraph tests and complex graph queries

The test returns either the Name of such a configuration, an *arbitrary* one if several configurations do exist in parallel, or *fails*. It is used within ConsistentConfiguration for retrieving any minimal but complete configuration, where imports of included variants are satisfied by included variants in turn. This additional condition is guaranteed by the negation of the second test UnresolvedImportExists. It contains an example of a *negative node pattern*, a crossed-out node (an example for a negative edge pattern, a crossed-out edge, is part of fig. 14 in chapter 6). It succeeds if matches for all *positive node and edge patterns* may be found such that a match for the negative node pattern does not exist. More precisely, it tests whether a configuration with name CName contains a VARIANT which imports a MODULE which has *not* a VARIANT which is part of the selected configuration. The complex import constraint is modeled as a *positive path condition*, the double arrow between nodes `1 and `3 with text label needs (cf. fig. 11 for the definition of needs).

The last test of fig. 9 checks whether a given configuration contains useless variants based on the assumption that all those module variants are useless which are not directly or indirectly needed by the main module's selected implementation variant. This constraint is expressed by a *negative path condition*, the crossed-out double arrow between nodes `4 and `6 with text label needs*. It requires that the match for node `6 is *not in the transitive closure* of needs starting at the match for node `4.

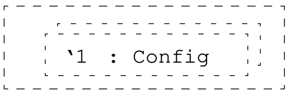
The *execution of the query* ConsistentConfiguration itself proceeds as follows: The first test returns the name of a possibly incomplete configuration. Afterwards (the symbol & represents sequential composition of calls), it checks that the selected configuration is complete and does not contain unneeded variants. If these checks succeed then the query *succeeds* as a whole and returns the name of the configuration. Otherwise, *backtracking starts*, reenters the first test call, and reactivates its pattern matching process. As a result, the first test either determines another configuration or fails. In the first case, execution of the query is continued with checking consistency of a new configuration; in the second case, the whole query fails and triggers backtracking or abortion of its calling query/transaction in turn.

In such a way it is possible to program very complex graph queries with PROGRES. Subgraph tests offer a powerful *visual sublanguage* for defining basic queries in a similar style as visual database query languages do (cf. [1, 2, 5]). Queries themselves with their complex control structures support a *relational style of programming* which is rather similar to the style of programming used within Prolog-like languages:

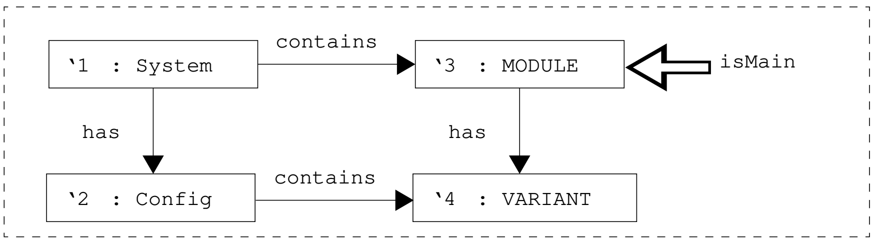
- The operator & corresponds to the *left/right-evaluation* of Prolog clauses.
- Another control structure <u>choose</u> ... <u>else</u> ... <u>end</u> corresponds to the *top/down selection* of Prolog clauses with matching heads.
- Furthermore, operators <u>and</u> and <u>or</u> are available which represent the proper logical $\wedge$ and $\vee$ and *evaluate* their subexpressions in a *randomly selected order*.
- Finally, *recursion* or *conditional iteration* as a shorthand for tail recursion can be used to specify even more complex graph queries.

Beside the relational style of programming, a completely *imperative style of programming* is supported. Fig. 10 contains a deterministic reformulation of the query from fig. 9, which makes in addition less usage of double negations and implicit existential quantification. The old subquery for "not exists not resolved import" is replaced by an inner <u>for all</u> loop, which tests whether all imports of all variants in a configuration have also variants in the configuration. Furthermore, nondeterministic assignments to the LocalName variable in fig. 9 together with backtracking is replaced by an outer <u>for all</u> loop, which determines first all existing configurations and selects then the subset of consistent configurations.

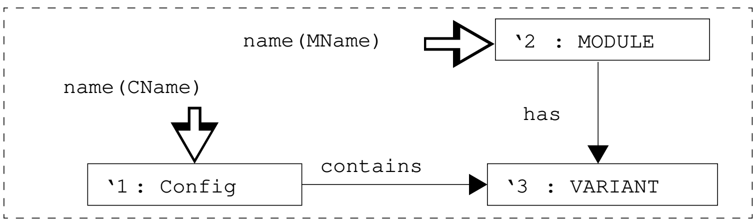
```
query AllConsistentConfigurations( out CNameSet : string [0:n] ) =
   use LocalNameSet, ResultNameSet : string [0:n] do
       ResultNameSet := nil             (* Start with the empty result set. *)
     & GetAllConfigurations( out LocalNameSet ) (* Get all 'Config' nodes. *)
     & for all LocalCName := elem( LocalNameSet ) do
                    (* Loop through all configurations and check the same    *)
                    (* consistency conditions as in ConsistentConfiguration. *)
         choose
           when ConfigurationWithMain( out LocalCName )
            and not ConfigurationWithUselessVariant( LocalCName )
            and for all LocalMName := elem( LocalCName.-has->.=needs=> ) do
                          (* Tests for all 'Variants' which are the targets  *)
                          (* of 'has' edges of a selected configuration:     *)
                          (*   do all needed modules have variants which are *)
                          (*   also part of the selected configuration?      *)
                        ModuleInConfiguration( LocalCName, LocalMName )
                end
           then ResultNameSet := ResultNameSet or LocalCName
                             (* Includes new configuration name into result. *)
         else
           skip          (* Else branch does nothing and may be omitted. *)
         end
       end
    & CNameSet := ResultNameSet       (* Returns all correct configurations *)
   end
end;
test GetAllConfigurations( out CName : string [0:n] )  =
```




```
   return CName := `1.Name;
end;
test ConfigurationWithMain( CName : string )  =
```




```
   condition CName = `2.Name;
end;
test ModuleInConfiguration( CName : string ; MName : string ) =
```




```
end;
```

Fig. 10: Replacing implicit backtracking in queries by explicit control structures

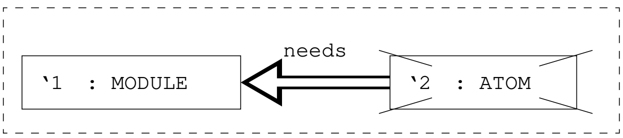
Until now, we did not explain the purpose of so-called *restrictions* and *path declarations* which were used within fig. 9, but which are declared in fig. 11. They are the means for defining derived sets of nodes with certain properties and derived binary relationships. Both restrictions and path expressions are used within attribute evaluation rules to *determine context nodes* with needed attribute values and within subgraph tests and productions as complex *application conditions*. The restriction `isMain` determines for instance the set of all those `MODULE` nodes which are not needed (imported) by other module interfaces or realizations (for reasons of simplicity we do assume that this condition characterizes main program modules sufficiently). It is true for all those `MODULE` nodes which are not reachable from another `ATOM` node via a separately defined path `needs` (expressed by the double arrow with label `needs`). The following restriction `name` is used as an abbrevation for the often needed selection of a `UNIT` node with a particular name.

The last two declarations of fig. 11 define derived binary relations in the form of complex *path expressions*. Path expressions are primarily relational definitions of derived relationships. Nevertheless, they may also be understood as functions, which navigate from a given set of source nodes along certain edges in a host graph and return their target nodes. The notion of a *path expression* within the area of graph grammars was originally introduced to determine and manipulate the embedding of matched subgraphs in their host graph (cf. following section 6). They are nowadays more important for defining complex application conditions of subgraph tests and graph rewrite rules. Therefore, our notion of path expressions is almost equivalent to the notion of path expressions as it used within the database community together with object-oriented or ER-like data models (e.g. in [14]).

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

- *edge operators* of the form `-e->` and `<-e-` allow the traversal of `e`-labled edges from source to sink or vice versa,
- *attribute conditions* of the form `valid expr`, as used within the restriction `name`, require that a given node on a path has certain attribute values,

```
restriction isMain : MODULE =
  `1 in
```




```
end;

restriction name ( UnitName : string ) : UNIT =
   valid (self.Name = UnitName)
end;

path needs : ATOM [0:n] -> MODULE [0:n] =
   (  ( instance of VARIANT &
      ( -v_uses-> or ( <-has- & instance of MODULE & -m_uses-> ) ) )
   or ( instance of MODULE & -m_uses-> )
    )
end;

path dependsOn : MODULE [0:n] -> MODULE [0:n] =
   ( self or -has-> ) & =needs=>+
end;
```

Fig. 11: Specification of restrictions and paths (derived sets and relations)

- *context restrictions* of the form `with path` require that a given node n is related via a path expression `path` to at least one node m,
- and *node type restrictions* of the form `instance of type_set` require that the type of a node is an element of the given type set.

Many different *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, and finally `p+` (or `p*`) for computing the transitive (and reflexive) *closure* of a given path expression `p`. These constructs are presented within fig. 11. The path `needs`, for instance, computes the union of the following to subexpressions:

- The first subexpression returns for any `ATOM` node n, which is an instance of (a type of class) `VARIANT`, the union of another two subexpressions. The first one consists of all targets of `v_uses` edges which have n as their source; the second one computes a set S of all `MODULE` instance sources of `has` edges, which have n as their target, and returns then the set of all `m_uses` edge targets, which have a node in S as their source.
- The second subexpression returns for any `ATOM` node n, which is an instance of (a type of class) `MODULE`, the set of all targets of `m_uses` edges, which have n as their source.

As a result, the relationship { `(6, 3)`, `(6, 4)`, `(3, 4)`, `(7, 4)` } is the semantics of the path declaration `needs` for the graph of fig. 4. The semantics of the following path declaration `dependsOn`, the *transitive closure* of `needs` applied to a given module node and all its variants, is the relationship { `(2, 3)`, `(2, 4)`, `(3, 4)` }.

Please note that the presented examples do not make usage of all existing path operators. Additionally available operators are offered for computing *intersections* (`p1 and p2`) or *differences* (`p1 but not p2`) of intermediate result sets as well as for conditional branching and iteration:

`[ not with -m_uses-> :: -has-> & -v_uses-> | -m_uses-> ]`

is a more or less meaningless example for *conditional branching*. It returns for a given node n the result of `-has-> & -v_uses->` if n is not the source of an `m_uses`-edge, it returns the result of `-m_uses->` otherwise. An equivalent shorthand without an explicitly mentioned condition (before `?`) is

`[ -m_uses-> | -has-> & -v_uses->`].

Replacing the square brackets [and] of any conditional path expression by curly braces { and } we get an example for *conditional iteration* of path expressions:

`{ -m_uses-> | -has-> & -v_uses->` }

iterates as long as one of its two branches returns a nonempty result with a preference for results of the first branch. Its application to the node `2` of fig. 4 is computed as follows:

- The evaluation of the first branch fails but the evaluation of the second branch returns the intermediate result { `3, 4` }.
- The final result is now the union of the application of the iteration construct to any node of the previously computed intermediate result.
- The evaluation with `4` as input fails for both branches and returns, therefore, the node `4` itself.
- The evaluation with `3` as input results in another intermediate result set { `4` } which is already an element of a maintained set of already visited nodes and, therefore, no longer of any importance for the final result set.
- The final result of the evaluation process is than the result of step (c), the set { `4` }.

Please note that *conditional iteration* has a rather different behavior than transitive closure; it returns not the set of all visited nodes, but only those nodes which are end points of the iteration process. Furthermore, its *termination* even *in the presence of cycles* is guranteed by maintaining an internal set of already visited nodes. This has the consequence that conditional iteration never returns its input nodes, except the case where the first iteration step fails. Let us assume for instance that the graph of fig. 4 contains an additional `m_uses`-edge from `3` to `2`. In this case, the evaluation of our conditional path expression returns the same result as before, the set `{ 4 }`.

To summarize, all *visual constructs* available for the definition of subgraph tests and many *textual control structures* (similar to those used for the definition of complex queries or attribute expressions) can be used for the declaration of paths and restrictions. The textual style of programming should only be preferred when transitive closures are needed or when alternatives must be considered. Finally, we have to mention that it is a matter of additional pragmas whether a restriction or a path is recomputed every time when it is needed or whether its results are *materialized* once and *incrementally maintained* in a consistent state using an extension of the afore-mentioned attribute evaluation mechanism (cf. section 7).

6. Definition of Graph Transformations

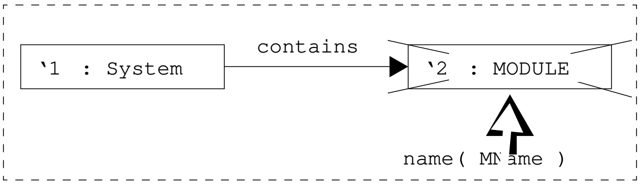
Subgraph tests and queries together with path declarations and restrictions were the main constructs for inspecting already existing graphs, whereas productions and transactions are their counterparts for creating and modifying schema consistent graphs. *Productions* have a *left*- and a *right-hand side graph pattern* as their main components, where left-hand side patterns are the same as already presented subgraph patterns of tests. *Transactions* use the same control structures to construct complex graph transformation processes from basic production applications in the same way as queries use them to construct complex graph analysis processes from basic subgraph tests.

The first production of fig. 12 selects the `System` node and verifies that a module with the name `MName` is not already part of our software design. In that case, the match of its left-hand side is *replaced* by an image of its right-hand side in the following manner (note that we will use the words "node/edge" not only in their usual sense but also as an abbreviation for "image or match of node/edge pattern in host graph" in the sequel):

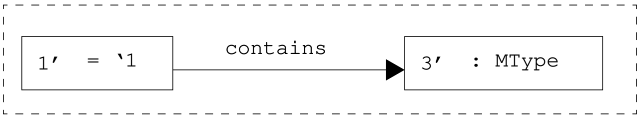
- All nodes in the right-hand side which are bound to nodes of the left-hand side are *preserved* without performing any not explicitly required intrinsic attribute value or edge context modifications. This means in our case that the matched `System` node `` `1 `` is preserved.
- All positive nodes and edges of the left-hand side which have no counterparts in the right-hand side have to be *deleted*, including all incident edges of deleted nodes. The left-hand side of our example contains only an identically replaced node `` `1 ``, a negative node `` `2 `` (which must not exist), and an implicitly negative edge from the positive node `` `1 `` to the negative node `` `2 ``. Therefore, nothing is deleted at all.
- All nodes and edges of the right-hand side with no counterparts in the left-hand side are *added* to the host graph, i.e. a new node of type `MType` is created which is connected to the already existing `System` node via a new `contains`-edge.
- Finally, *new attribute values* are computed by evaluating expressions which may reference input parameters as well as old attribute values (these expressions are computed before any modification of the host graph is performed).

This example illustrates the possibilities of PROGRES to define *parametrized productions* which must be instantiated (in the sense of a procedure call) with actual attribute values and node types. In this way, a single production may abstract from a set of productios which differ only with respect to used attribute values and types of matched or created nodes. In almost all cases, node type parameters are not used for matching purposes, but provide concrete types for new nodes of the right-hand side.

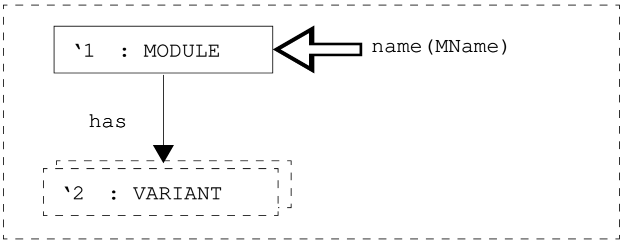
```
production CreateModule ( MName : string; InterfaceDescription : file;
                          MType : type in MODULE ) =
```



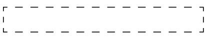

```
::=
```



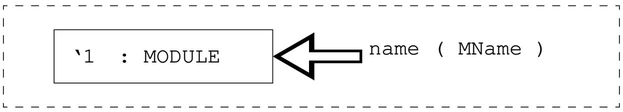

```
   transfer 2'.Name := MName;
            2'.File := InterfaceDescription;
end;

production DeleteModule ( MName : string ) =
```



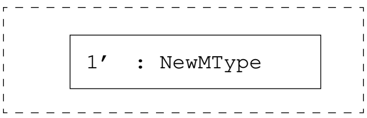

```
::=
```



```
end;

production ChangeModuleType ( MName : string; NewMType : type in MODULE ) =
```




```
::=
```




```
   embedding redirect  -m_uses->, -has->, <-contains-,
                      <-m_uses-, <-v_uses- from '1 to 1';
   transfer 1'.Name := '1.Name;
            1'.File := '1.File;
end;
```

Fig. 12: Specification of basic graph transformations

The replacement of the node inscription `'2 : MODULE` in the left-hand side of production `CreateModule` by the inscription `'2 : MType` would change the production's semantics considerably. The old inscription prohibits the existence of a node of *any type* of the class `MODULE` with the same name, whereas the new inscription would only prohibit the existence of a node of *exactly the same type* of the class `MODULE` as the node we are going to create. It would then be possible to create modules of different types which, however, have the same name.

Note that the class `MODULE` is used to denote a *type for the node type parameter* `MType`. Having this information around we are then able to verify that any application of the production `CreateModule` to a *schema consistent graph* produces another schema consistent graph, i.e. that the new node is always a legal target of a `contains`-edge and that it carries indeed a `Name` attribute (of type string). The following lines would produce all three modules of our sample architecture in fig. 3.1 (without their import relationships):

```
CreateModule("Main", TextFile1, FunctionModule) &
CreateModule("UserInterface", TextFile2, ADOModule) &
CreateModule("Files", TextFile3, ADTModule)
```

The following production `DeleteModule` of fig. 12 removes a specified `Module` node from a MIL host graph together with all its `VARIANT` nodes. The dashed double borders of the node `'2` indicate that it may not only be bound to a single node in the host graph, but to the *maximum set of all nodes* fullfulling all required constraints (they must be the targets of `has`-edges emanating from node `'1`). In the same way as we had four different options for edge type and variable/parameter cardinalities, *four different kinds of node patterns* are available:

- Solid simple boxes are *obligate node patterns* which match single node instances in the host graph,
- dashed simple boxes are *optional node patterns* which match a single node with the required properties if existent but do not cause failure of the overall matching process if such a node does not exist,
- dashed double boxes are *optional node set patterns* which match a possibly empty set of nodes such that any node of the given set fulfills all mentioned requirements, and
- solid double boxes are *obligate node set patterns* which match a nonempty set of nodes and cause failures, where dashed (double) boxes match the empty set.

Taking all these additional kinds of node patterns into account, we have to extend the overall *proceeding for finding matches* of left-hand sides. The first step is to find matches for all dashed or solid *simple node patterns and edge patterns* such that all positive constraints in the left-hand side are taken into account and no two node patterns match the same node in the host graph (isomorphic subgraph matching). The next step then is to find maximal matches for all *node set patterns* such that no two node (set) patterns match the same node in the host graph. Finally, one *negative node or edge pattern* after the other is handled by trying to extend the already determined subgraph with a match of the negative node or edge. Again, we have to ensure that (negative) node patterns do not match nodes which are already part of the selected subgraph. Any existing extension causes failure and initiates backtracking, which tries to find another match within the host graph.

The above-mentioned restriction to *isomorphic subgraph matching* can be overwritten. If two node (set) patterns with identifiers `'x` and `'y` may (but need not) share (a subset of) their matches then an explicit *folding clause* of the form `fold 'x, 'y;` has to be added to a production. For further details concerning (nonisomorphic) pattern matching and the treatment of positive as well as negative application conditions the reader is referred to [46]. For further details concerning the problem to *find matches efficiently*, the reader is referred to [45]. There, we shall discuss the implementation of pattern matching briefly.

Using identical replacement of nodes or node sets together with the options to allow nonisomorphic matches and to connect nodes in the left-hand side via arbitrarily complex path expressions, offers a PROGRES user powerful means to construct and to modify so-called *embeddings*. The general problem, postponed until now, is that we have to create or modify connections between the unmodified rest of the host graph and the new nodes or the identically replaced nodes. There exist *two strategies* for solving this problem in graph grammar literature. The first one is based on *identical replacement* of nodes and was used in the first two productions of fig. 12. Its "traditional" form became popular with the so-called algebraic graph grammar approach, where it is restricted to identical replacement of single nodes (cf. [7]). PROGRES, in the contrary, allows for the identical replacement of arbitrarily large sets of nodes.

Using the identical replacement strategy, when many different types of edges have to be redirected, results in unreadably large left- and right-hand sides with many set node patterns and folding clauses (folding clauses are necessary when two embedding edges may have the same context node). Therefore, PROGRES offers also the second embedding strategy based on *additional embedding clauses* and path expressions, which became popular with the algorithmic (set-theoretic) branch of graph grammars (cf. [30]).

The production `ChangeModuleType` of fig. 12 contains an example of an embedding clause. The notation

```
redirect e1, e2,... from 'n to m';
```

is a shorthand for

```
redirect e1 from 'n to m';redirect e2 from 'n to m'; ...
```

and

```
redirect -x-> from 'n to m';
```

is in turn a shorthand for

```
remove -x-> from 'n; (* not necessary for deleted 'n *)
copy -x-> as -x-> from 'n to m';
```

It affects any `x`-edge which connects before the rewriting step the match of `'n` to another (not matched) context node v. It replaces it by another `x`-edge which connects the match of the (new) node `m'` to the context node v after the rewrite step. Using `<-y-` instead of `-x->` at certain places within the lines above, it is also possible to change the type or direction of a whole bundle of edges, which leave the match of the left-hand side or enter it, by a single embedding rule.

Both embedding strategies have their specific (dis-)advantages. The first one, based on identical replacement and node set patterns, has a *graphical notation* and is more readable if edges of a single type have to be redirected and if additional conditions about context node types etc. have to be considered. The second one, based on embedding clauses and path expressions, has a very compact *textual notation*; it is preferable if edges of many different types may or may not exist, which simply have to be redirected.

Returning from these general remarks about the embedding problem, we apply the following sequence of productions

```
DeleteModule("Files") &
CreateModule("FileSystem", TextFile, ADOModule) &
ChangeModuleType("UserInterface", FunctionModule)
```

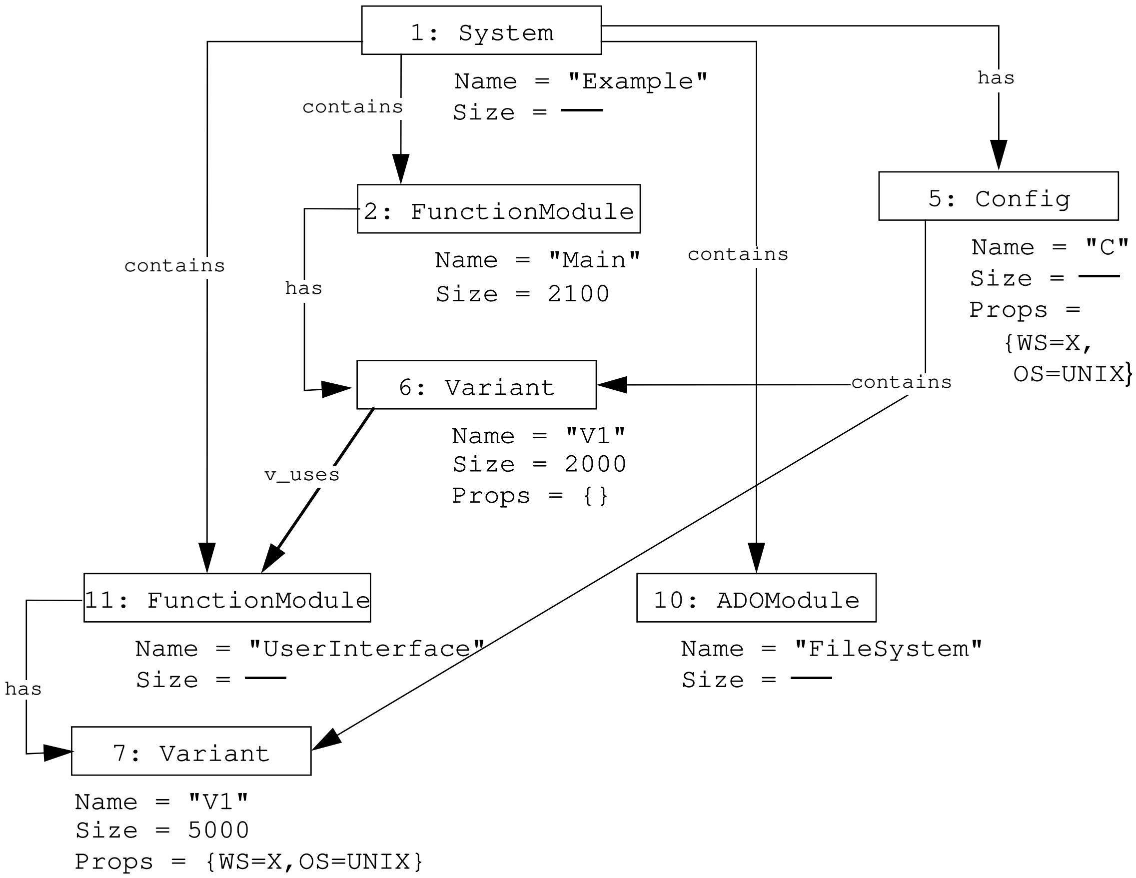


Fig. 13: A modified "MIL" graph derived from the previously presented MIL graph

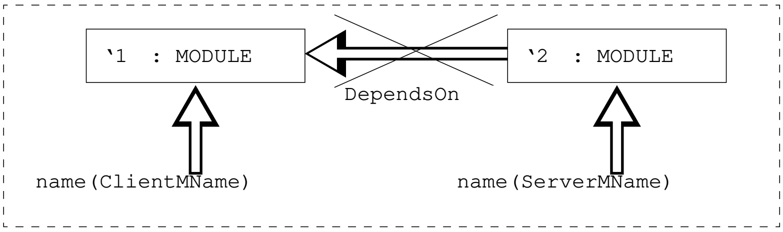
to the graph of fig. 4. This results in another MIL graph which is displayed in fig. 13 and computed as follows:

- The first production call deletes the node `4` including its `Variant` nodes `8` and `9` as well as all incident edges of these three nodes.
- The next production call creates a new node `10` which is connected via a `contains`-edge to the old `System` node `1`.
- The last call deletes the `ADOModule 3` with all its incident edges, creates a new `FunctionModule` node `11`, and redirects all previously existing connections of the deleted node to the new node.

Note that the figure does not display any `File` attributes and uses "—" to mark all possibly affected values of derived `Size` attributes, which must be recomputed when needed.

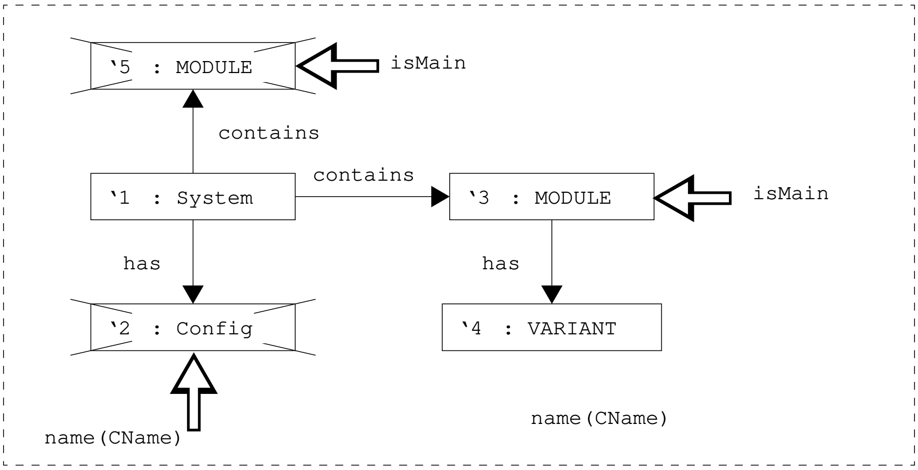
Until now, we did not use any *complex application condition*s within productions in the form of path expressions or attribute expressions (but we know them from the definition of subgraph tests). The following fig. 14 contains examples for these features of the language PROGRES. The first production `CreateMUse` creates a single edge of type `m_uses` and deletes nothing at all. This edge represents an import from a client module `ClientMName` to a server module `ServerMName`. The crossed out double arrow (negative path expression) with label `dependsOn` (cf. fig. 11) prohibits the application of the production, if the server module already depends, directly or indirectly, on the client module. In this case, the creation of cyclic (interface) import chains is prohibited.

```
production CreateMUse( ClientMName, ServerMName : string ) =
```

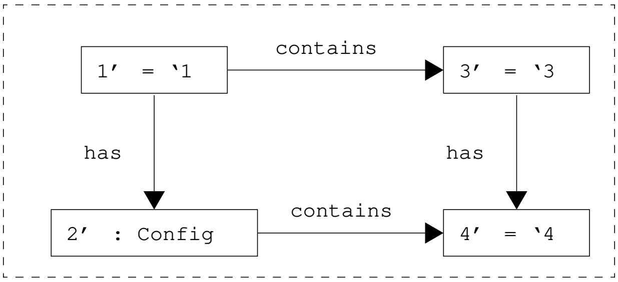



```
::=
```


```
end;

production InitConfig(    CName : string;
                            ReqProps : string [0:n];
                        out NewProps : string [0:n] ) =
```




```
::=
```

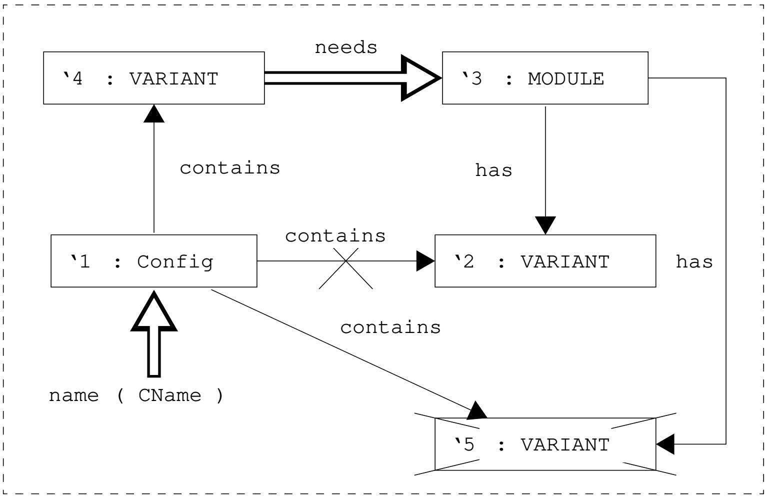



```
   condition  ‘4.Props ‘are_in’ ReqProps;
   transfer   2’.Name := CName;
              2’.Props := ReqProps;
   return     NewProps := merge( ReqProps, ‘4.Props );
end;
```

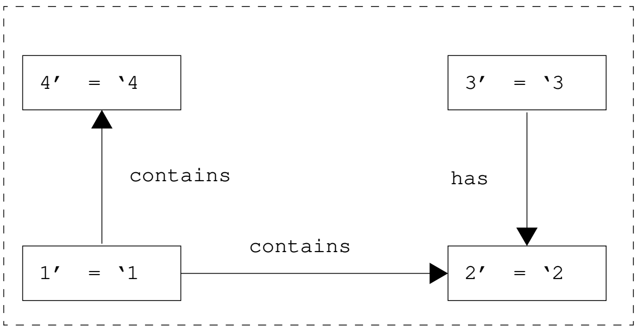
Fig. 14: Specification of additionally needed complex productions

The following production `InitConfig` implements the first step of the configuration of a system variant, which fulfills the given set of requirements `ReqProps`. It requires that a software system has a uniquely determined main module, which has in turn at least one variant fullfilling the initial set of requirements. The *complex attribute condition* is expressed using the function `'are_in'` (any binary/unary function with a name of the form `'...'` or a name which consists of special symbols like `!*#$$%^&*+-=<>?/` only may be called using a infix/prefix notation). It tests whether all required properties in `ReqProps` are covered by the set of guaranteed properties `'4.Props` of the regarded module variant. Its specification is rather similar to the specification of the function `merge` of fig. 6 and, therefore, omitted over here.

```
production ResolveImport (    CName : string; ReqProps : string [0:n];
                          out NewProps : string [0:n] ) =
```




```
  ::=
```




```
   condition '2.Props 'are_in' ReqProps;
   return NewProps :=
end;

transaction CreateConfig( CName : string; CProps : string [0:n] ) =
   use ReqProps := CProps
   do
       InitConfig( CName, ReqProps, out ReqProps )
     & loop
          ResolveImport( CName, ReqProps, out ReqProps )
       end
     & not UnresolvedImportExists( CName )
   end
end;
```

Fig. 15: The main graph transformation for producing a configuration

A selected variant may have additional requirements concerning its runtime platform. These additional requirements are merged with the original set of requirements and returned as an *out parameter* to a calling transaction. In such a way it is possible to take additional restrictions for the rest of the configuration process into account.

The second needed production `ResolveImport` for producing a *consistent and complete configuration* is shown in fig. 15. It looks for an already included `VARIANT` node with import edges from another `MODULE` node which is not already part of the configuration. Any `VARIANT` of this node, which fulfills all already assembled requirements, will be included into the configuration. Again, a set of new requirements is built from the old set of requirements and the set of requirements of the new included `VARIANT`.

The productions `InitConfig` and `ResolveImport` together with the previously introduced test `UnresolvedImportExists` are the *basics blocks* of the overall system configuration process. Building such a configuration is a typical software document analyzing and modifying operation, which cannot be specified within a single graph rewriting production. To execute the corresponding graph transformation, many basic graph rewriting steps are necessary, which have to be *executed in a particular order*. Therefore, PROGRES offers the same *control structure concepts* for the definition of complex graph rewriting transactions, which were already presented above together with the discussion of queries in section 5.

The transaction `CreateConfig` (cf. fig. 14) needs the *sequential composition operator* `&`, the *negation* `not` of a test, and the *conditional iteration* `loop ... end`. The hidden termination condition of this loop is that its body is no longer executable. It could be written down explicitly as follows:

```
loop when def( ResolveImport( ... ) then
   ResolveImport( ... )
end
```

The used `def` operator has about the same semantics as the `with` operator for path expressions. It tests whether the execution of its argument would be successful or not without modifying the host graph (all temporary modifications during the execution of its argument are cancelled afterwards using backtracking). It is a general rule that the conditions of branching or iterating control structures, also called *guards*, *may not have any side effects*. This is guaranteed by the PROGRES language's static semantics rules.

Other control structures support conditional deterministic branching (see below), *nondeterministic branching* (`t or t′` applies either `t` or `t′` to a graph), and *nondeterministic sequential composition* (`t and t′` is equivalent to `(t & t′) or (t′ & t)`, i.e. executes both `t` and `t′` in an arbitrarily selected order). It would, for instance, be possible to replace the above explained conditional iteration by a call to the *recursively defined transaction*

```
transaction Iter( ... ) =
   choose
      ResolveImport( ... ) & Iter
   else
      skip
   end
end;
```

which shows an example for *conditional deterministic branching*. The condition of the choose statement's branches are again hidden and could be added as follows:

```
transaction Iter( ... ) =
   choose
      when def( ResolveImport( ... ) & Iter )
      then ResolveImport( ... ) & Iter
   else
      when def( skip ) (* condition is always true *)
      skip (* succeeds always without any effects  *)
   end
end;
```

For further details concerning these control structures see [44, 46].

The main advantage of our control structures in comparison to other programmed graph rewriting approaches, as for instance [6, 18], is that whole transactions have the same characteristics as single productions:

- They are *atomic* with respect to their effects onto a given host graph. Any execution attempt either succeeds as a whole or aborts without any graph modifications.
- They are *consistency preserving*, i.e. manipulate schema consistent graphs only.
- They make their own *nondeterministic choices* and initiate *backtracking* when a particular decision leads into a dead-end later on.

As a consequence, transactions over here fulfill (more or less) the well-known *ACID properties of database transactions*: atomicity, consistency, isolation, and duration. In the single user PROGRES environment, the principle of *isolation*, which requires that the parallel execution of a set of transactions has the same effects as their serialized execution, makes no sense. Generated stand-alone prototypes of specified tools fulfill this requirement in a rudimentary way by locking the whole host graph for the duration of a top-level transaction.

The last property, *duration* of the effects of a completed transaction, is guaranteed for top-level transactions, which are called from an “external” client. This is irrelevant in the single user PROGRES environment, but it is in turn important for generated tool prototypes. In this case, backtracking before “commit” points of top-level transactions is no longer useful and the still available ability of the basic execution machinery for undoing effects of any kind of transactions is only offered in the form of explicitly activated undo (and redo) commands (see also [26] about the underlying database system GRAS).

The execution of the presented transaction `CreateConfig` (with an empty set of requirements) applied to the graph of 4 (without the `Config` node `5` which will be created now) may, therefore, proceed as follows:

- The tests whether the constructed configuration is complete. This is appearently not the case such that the called test succeeds and its negation fails. This initiates backtracking.
- The execution returns then to the last choice point of the ongoing execution process and cancels all immediately executed graph transformations and variable or parameter assignments.
- As a result the second call to the production `ResolveImport` is reactivated and another remaining match is selected instead of the previous one. This leads to the selection of the `"UNIX"` variant of the module `"UserInterface"`.
- Afterwards, the third call to `ResolveImport` succeeds and selects the only existing variant of the third needed module.
- Finally, the loop terminates again and the previously failed completeness test is now successfull, thereby allowing a “commit” of the whole transaction execution process.

7. Advanced Features

Within previous sections we did not talk about *efficiency problems* and we developed a new specification from scratch, instead of *reusing* already existing *parametrized specification fragments*. Both topics are not relevant as long as PROGRES is used as a pure specification language for toy examples. But these aspects cannot be neglected any longer as soon as the language is used to describe and prototype "real" software engineering tools (as promised at the beginning of paper). Then additional and very recently added language constructs are needed. They are

- *pragmas* within graph schemata which have significant influence onto the runtime behavior of pattern matching strategies,
- formal *node parameters* by means of which information about already found subgraph matches may be passed from one rewrite rule to the next one,
- *meta attributes* which are attributes of node types instead of node instances and which allow for the definition of parametrized graph schemata, and
- advanced forms of *parametric polymorphism* which are needed to specify highly parametrized graph transformation processes.

These language constructs are the topic of this chapter and are of real importance as soon *efficient and reusable subspecifications* have to be built. We will start with the discussion of efficiency matters and continue then with the definition of parametrized graph schemata and graph transformations.

The transaction `Test` of fig. 16 is inefficient with respect to its runtime behavior due to the following reasons:

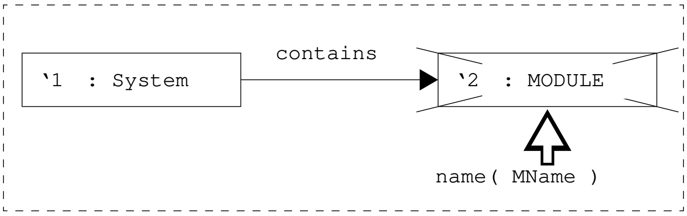
- That part of production `CreateModule` which guarantees that a software system does not already contain a module with name `MName` computes first the set of all `MODULE` nodes of the selected `System` node and checks then for any node in this set the restriction `name(MName)` of fig. 11.
- The following production `CreateMUse` loops even twice through the set of all `MODULE` nodes for finding those nodes which have the name `ClientMName` and `ServerMName`.
- Even worse, the knowledge of the first two `CreateModule` production calls of transaction `Test` about the nodes with name `"UI"` and `"Files"` is not used in the third production call.
- Finally, the transitive closure of the path `dependsOn` with the match of node pattern `` `2 `` as input is recomputed within any call to the production `CreateMUse`.

It is a matter of debate whether the last point (d) above is really a source of inefficiencies. Any successfull call to the production `CreateMUse` modifies indeed the derived relation `dependsOn`. Therefore, it might be more efficient to compute any application of `dependsOn` to a specific node from scratch instead of maintaining and reusing previously computed results. In the latter case, we have the additional overhead for invalidating and recomputing inconsistent results. But it is beyond any doubts that the "materialization" of the results of the restriction `isMain` and the path `needs` of fig. 11 would accelerate the execution of the transaction `CreateConfig` of fig. 15 considerably. In this case, their results are needed again and again within called productions but are not affected at all.

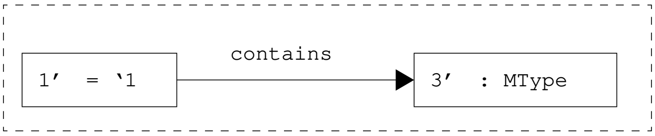
The fig. 17 contains a specification fragment, where all above mentioned problems are eliminated. All modified pieces of "code" are highlighted in boldface:

- The *pragma* index within the declaration of attribute Name causes the creation of an *attribute index* which allows for the efficient computation of the set of all UNIT nodes with a distinct name. Using the index, the loop over all MODULE nodes within production CreateModule is no longer necessary (another pragma key should be used if different UNIT nodes with the same Name value may not exist).
- The *pragma* static before the restriction isMain and the paths needs and dependsOn causes their *materialization*. An extended version of the incremental attribute evaluation algorithm is used to maintain materialized results in a consistent state. Materialized (parameterless) restrictions are explicitely stored node sets with certain properties, whereas materialized (parameterless) paths correspond to explicitely stored derived binary relations.

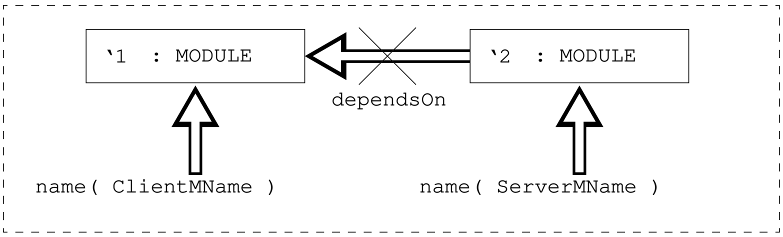
```
production CreateModule( MName : string; InterfaceDescription : file;
                         MType : type in MODULE ) =
```



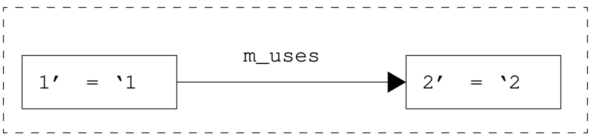

```
   ::=
```




```
   transfer 2'.Name := MName;
            2'.File := InterfaceDescription;
end;

production CreateMUse( ClientMName, ServerMName : string ) =
```



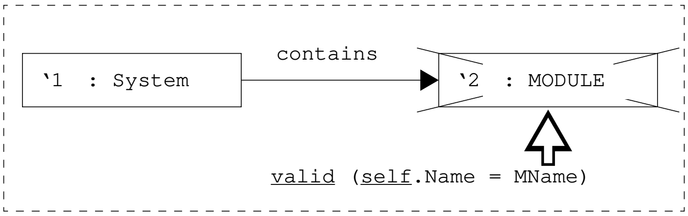

```
         ::=
```



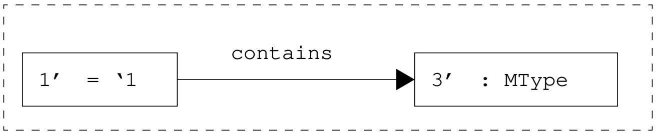

```
end;

transaction Test =
    ...
  & CreateModule ( "UI", UITextFile, ADOModule )
  & CreateModule ( "Files", FilesTextFile, ADTModule )
  & CreateMUse ( "UI", "Files" )
  & ...
end;
```

Fig. 16: Highly inefficient specification of graph transformation

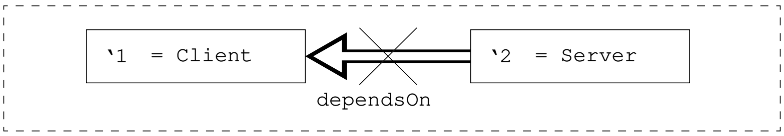
```
node class UNIT
   ...
   intrinsic index Name : string;
end;

  ...

static restriction isMain : MODULE = ... end;

static path needs : ATOM [0:n] -> MODULE [0:n] = ... end;

static path dependsOn : ATOM [0:n] -> MODULE [0:n] = ... end;

production CreateModule(   MName : string;
                             InterfaceDescription : file;
                             MType : type in MODULE;
                         out NewModule: MODULE ) =
```



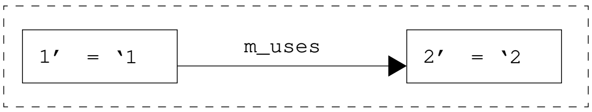

```
   ::=
```




```
   transfer 2'.Name := MName;
            2'.File := InterfaceDescription;
   return   NewModule := 3';
end;

production CreateMUse( Client, Server : MODULE ) =
```




```
      ::=
```




```
end;

transaction Test =
    ...
  & use UIModule, FilesModule : MODULE do
      & CreateModule( "UI", UITextFile, ADOModule, out UIModule )
      & CreateModule( "Files", FilesTextFile, ADTModule, out FilesModule )
      & CreateMUse( UIModule, FilesModule )
      & ...
    end
  & ...
end;
```

Fig. 17: Runtime improving modifications of a specification fragment

- Finally, the pattern matching process of the production `CreateMUse` is considerably improved by *providing the production with matches* for the node patterns in its left-hand side. For this purpose, the production `CreateModule` returns *handles* to created `MODULE` nodes in its out-parameter `NewModule`. These handles are stored within the local variables `UIModule` and `FilesModule` and then used as in-parameter values for production `CreateMUse`.

These *improvements* have the effect that the pattern matching process for the production `CreateModule` accesses first an always existing index for all `System` nodes (which is always a singleton set in the running example), accesses then the index for the attribute `Name` (which returns in our case at most one `MODULE` node), and checks finally the existence of a `contains` edge between retrieved nodes. The pattern matching process for the production `CreateMUse` is even simpler. It has just to check whether the provided nodes `Client` and `Server` module are connected via a materialized `dependsOn`-edge.

In such a way, it is possible to write quite *efficiently executable* graph transformations *except the considerable overhead* for storing graphs in a database, maintaining logs for backtracking (and recovery) purposes, and keeping indexes as well as derived data in a consistent state (on demand). It is the subject of ongoing research activities to develop new means for getting rid of unnecessary backtracking logs, cutting of useless execution paths of our depth-first search and backtracking strategy, and keeping graphs in main memory if required.

Moving from the discussion of efficiency matters onto the subject of *parametrized specification fragments* we will alter the decision that different types of modules do not have different types of variants. Therefore, we will replace the node type `Variant` of fig. 5 by three new node types `FunctionVariant`, `ADTVariant`, and `ADOVariant`. The type system of the current PROGRES version is not able to guarantee that `has`-edges connect only `MODULE` nodes and `VARIANT` nodes of corresponding types, i.e. that `FunctionModules` have only `FunctionVariants`, etc. It is again the subject of ongoing research activities to find out whether it is possible to extend the type system appropriately.

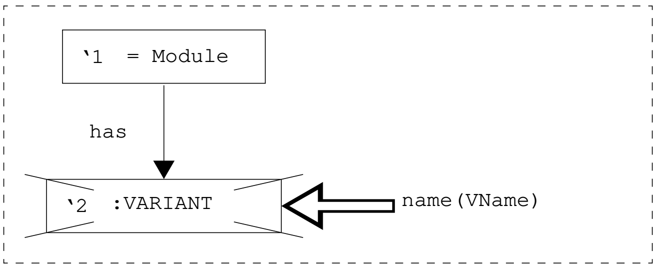
Currently, there are *two ways to circumvent the problem*. The first one is *type safe* and replaces the single edge type declaration `has` by three edge type declarations of the form

```
edge type xHas: xModule -> xVariant end;
```

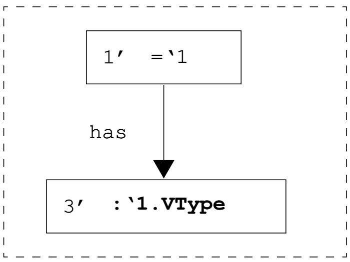
with `x` being either `Function` or `ADT` or `ADO`. This solution has the unwanted consequence that we need a seperate production for creating each type of `VARIANT` nodes (formal edge type parameter are not supported due to unsolved type checking problems of such a urgently needed language extension). Furthermore, any occurrence of the subexpression `-has->` within path expressions must be replaced by the new subexpression `-FunctionHas-> or -ADOHas-> or -ADTHas->` (subexpressions of the form `<-has-` must be treated analogously). Therefore, another solution should preferred which is *not type safe*, but which is much more elegant and compact. It uses so-called meta attributes to define a kind of *parametrized node classes* and parametric polymorphism with node and node type parameters to define *parametrized graph transformations*.

The specification fragment in fig. 18 contains a single production `CreateVariant` which creates for any existing type of `MODULE` a `VARIANT` node of the corresponding type. The correct `VARIANT` type for a given `MODULE` node is stored in its *meta attribute* `VariantType`. The value of such a meta attribute is defined within the graph schema (similar to derived attributes) and may not be modified at runtime (in contrast to intrinsic attributes). It has the same value for all nodes of a distinct type or, more precisely, it is not an attribute of a node instance but an *attribute of its node type*.

```
node class MODULE is a SPECIFICATION, ATOM
   ...
   meta
      VType: type in VARIANT;
   derived
      index Ok = for all Variant := self.-has-> ::
                      Variant.type = self.VType
                  end;
end;

node type ADOModule : MODULE
  redef meta
     VType = ADOVariant;
end;

   ...

node class VARIANT is a REALIZATION, ATOM end;

node type ADOVariant : VARIANT end;

   ...

production CreateModule(   MName : string; InterfaceDescription : file;
                             MType : type in MODULE;
                         out NewModule : MType ) =
   ...
end;

production CreateVariant( Module : MODULE;
                          VName  : string ; PSet : string [0:n] ) =
```



::=




```
   transfer 3'.Name := VName;
            3'.Props := PSet;
end;

transaction Test =
    ...
  & use UIModule : ADOModule do
      & CreateModule( "UI", UITextFile, ADOModule, out UIModule )
      & CreateVariant( UIModule, "V1", PSet )
      & ...
    end
  & ...
end;
```

Fig. 18: Meta node type attributes and parametric polymorphism

Assuming that `'1` matches a node of type `ADOModule`, the following equivalences hold:

`'1.VType = '1.type.VType = ADModule.VType = ADOVariant`

As a consequence, the new node pattern `3' : '1.VType` creates always a node of the correct `VARIANT` type.

Nevertheless, an additional derived attribute `Ok` is defined for demonstations purposes, which offers direct access to all modules wich have at least one variant of the wrong type (or which have only proper variants). It is again subject of ongoing research activities to extend the language PROGRES in such a way that it supports the definition of *complex integrity constraints in graph schemata* (not only via the workaround of using derived boolean attributes) and checks these integrity constraints automatically at runtime.

The declarations of fig. 18 resolve also another deficiency of the declarations of fig. 17. The formal parameter list of production `CreateModule` reveales now the fact that the returned `MODULE` node in the formal parameter `NewModule` has always the actual input value of the type parameter `MType` (the old type definition `MODULE` is replaced by a new type definition `MType`). As a consequence, it is also possible to replace the unspecific type `MODULE` of the local variable `UIModule` in transaction `Test` by the new type `ADOModule`. The type check of the `CreateModule` call in transaction `Test` proceeds now as follows:

- A *first pass* over the parameter list assigns each node type `in`-parameter the static type of its actual parameter. In our case, `MType` is bound to the node type `ADOModule`.
- A *second pass* checks then the compatibility of formal parameter types and actual parameter types by using the assembled information of the first pass. It replaces in our case the occurrence of `MType` within the declaration of `NewModule` by its actual type `ADOModule` and checks then that the type of the local variable `UIModule` is compatible with the type of the `out`-parameter `NewModule`.

This was just one example of “true” *parametric polyomorphism* in PROGRES. The general rule is that visible node type parameter identifiers may be used at any place, where node class or node type identifiers are allowed. A polymorphic path or function which returns for a given `System` node the set of all `MODULE` nodes of a specified type may for instance be defined as shown within fig 19. In this way it is possible to write highly parametrized specification fragments without loosing the *safeness of an almost statically typed language.* The word “almost” refers to the fact that potentially failing *type restrictions* are supported in various form (`instance of` clauses within expressions, node patterns within subgraph tests and productions, …). These constructs are a strict necessity for a pattern matching based language and should not be mixed up with unsafe *type casts* of programming languages like Modula-2 or C. Type casts never fail and enforce the interpretation of an element of an arbitrary type as an element of the given type, instead of checking whether a given element has the required type or not at runtime.

```
path module(MType : type in MODULE): System [1:1] -> MType [0:n] =
  -contains-> & CheckType(MType)
end;

restriction CheckType(MType : type in MODULE) : MType =
 instance of MType
end;

function myModules(MType: type in MODULE;
                     SystemNode : System ) -> MType [0:n] =
  self.-contains->.CheckType(MType)
end;
```

Fig. 19: Further examples of parametric polymorphism

8. Summary

Within this paper we presented the IPSEN specification language PROGRES and its model-oriented approach to the specification of software engineering tools or more precisely, the specification of their document processing acitivities. The approach, called *graph grammar engineering*, is *model-oriented* due to the fact that we used directed, attributed, node and edge labeled graphs to model complex document structures and programmed graph rewriting systems to specify operations on these documents in terms of their effects on their internal graph representation.

Having finished the implementation of the presented language version within the last months, we are now starting to evaluate first experiences with the language within various projects (cf. [2, 20]) and to develop the language's *programming-in-the-large part*. Some existing specifications are already about 80 pages long and require new means for their decomposition into separate, reusable, and encapsulated subspecifications. Therefore, the presented language version may be classified as the programming-in-the-small kernel of a *very high level programming language* which is a kind of

- *object-based database definition language* with graphical as well as textual constructs for the declaration of graph database schemata,
- *relational (logic-oriented) language* with various means for the definition of derived graph properties,
- *hybrid visual/textual database query language* which allows for the definition of rather complex subgraph patterns,
- *rule-oriented visual specification language* with support for the construction of parametrized graph rewrite rules (productions) with complex preconditions, and an
- *imperative/procedural programming language* with deterministic and nondeterministic control structures for programming complex graph transformation processes.

In summary, we provided the reader with a brief *introduction* to PROGRES. and demonstrated the very *basic principles of graph grammar engineering* by specifying a software design and configuration editor which are the following:

(1) *Model document instances* as graphs and study "*use cases*" of typical document querying or modifying operations.

(2) *Refine the graph model* step by step and add any information needed or produced by graph processing tools in the form of additional nodes, edges, and attributes.

(3) Identify sets of node objects which have the same properties (own attributes as well as possible relationships to other nodes): This leads to the d*efinition of node and edge types* as well as node attributes.

(4) *Distinguish* between *intrinsic and derived data* and find needed computation rules for all kinds of derived data as well as useful initialization rules for intrinsic attributes.

(5) *Identify* node types which share subsets of their properties and arrange them in the form of a *multiple inheritance hierarchy* (lattice) of node classes with node types as leafs.

(6) *Introduce* for any needed document transformation (query) a corresponding *graph processing transaction* (query).

(7) Keep control structures within transactions and queries as simple as possible and *specify reusable basic actions* in the form of parametrized subgraph tests and rewrite rules.

(8) Repeat the steps (a) through (g) if it is not important whether a specification is efficiently executable or not, otherwise continue with steps (i) through (l).

(9) *Refine* the initial *graph model* (the result of steps (a) through (e)) by adding cardinality constraints, index or key attribute pragmas, and by “tuning” evaluation strategies for derived data.
(10) *Reorganize* not efficiently executable *subgraph patterns* by breaking them down into smaller subpatterns and by passing information about already found subpatterns from one test/rewrite rule to the next one via node parameters.
(11) *Eliminate nondeterminism* on the level of control structures and especially within loops, whenever possible.
(12) Iterate the steps (a) through (j) as long as necessary.

We are aware of the fact that the sketched procedure is still rather naive. Therefore, more elaborate graph grammar engineering methods for specific application domains were already developed (cf. [10, 12, 46] for further details) and are still under development.

9. References

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

[2] Andries M., Engels G.: *Syntax and Semantics of Hybrid Database Languages*, in: [13], 19-36

[3] Bell B., Lewis C.: *ChemTrains: A Language for Creating Pictures*, in: [43], 188-195

[4] Claus V., Ehrig H., Rozenberg G.: *Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, LNCS 73, Springer Verlag (1979)

[5] Czejdo B., Elmasri R., Rusinkiewicz M., Embley D.W.: *A Graphical Data Manipulation Language for an Extended Entity-Relationship Model*, in: IEEE Computer, vol. 23, no. 3, IEEE Computer Society Press (1990), 26-37

[6] Dörr H.: *Efficient Graph Rewriting and its Implementation*, Ph.D. Thesis, FU Berlin , LNCS 922, Springer Verlag (1995)

[7] Ehrig H.: *Introduction to the Algebraic Theory of Graph Grammars (a Survey)*, in: Proc. Int. Workshop on Graph-Grammars and Their Application to Computer Science and Biology, LNCS 73, Springer Verlag (1979), 1-69

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

[9] Ehrig H., Kreowski H.-J., Rozenberg G. (eds.): *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Springer Verlag (1991)

[10] Engels G., Lewerentz C., Nagl M., Schäfer W., Schürr A.: *Building Integrated Software Development Environments Part I: Tool Specification*, in: acm Transactions on Software Engineering and Methodology, vol. 1, no. 2, acm Press (1992), 135-167

[11] Ehrig H., Nagl M., Rozenberg G., Rosenfeld A. (eds.): *Proc. 3rd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, LNCS 291, Springer Verlag (1987)

[12] Engels G., Schäfer W.: *Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment*, in Ehrig et al. (Eds.): Proc. TAPSOFT ’85, LNCS 186, Springer Verlag, 179-193

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

[14] Elmasri R., Wiederhold G.: *GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model*, in: Chen P.P. (ed.): *Entity-Relationship Approach to Information Modeling and Analysis*, Elsevier Science Publ. B.V. (1983). 49-7

[15] Furnas G.: *New Graphical Reasoning Models for Understanding Graphical Interfaces*, Proc. of CHI'91, 71-78

[16] Göttler H., Günther J., Nieskens G.: *Use Graph Grammars to Design CAD-Systems*, in /9/, 396-410

[17] Gemis M., Paredaens J., Thyssens I., Van den Bussche J.: *GOOD: A Graph-Oriented Object Database System*, in: Buneman P., Jajodia S. (eds.): *Proc. 1993 ACM SIGMOD Conf. on Management of Data*, SIGMOD RECORD, vol. 22, no. 2, acm Press (1993), 505-510

[18] Göttler H., Günther J., Nieskens G.: *Use Graph Grammars to Design CAD-Systems*, in [8], 396-410

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

[20] Heimann P., Joeris G., Krapp C.-A., Westfechtel B.: *A Programmed Graph Rewriting System for Software Process Management*, appears in: Corradini A., Montanari U. (eds.): Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop, Volterra (Pisa), Italy, August 1995, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier Science Publ. (1995)

[21] Horwitz S.: *Generating Language Based Editors: a Relationally-Attributed Approach*, Ph.D. Thesis, Cornell University, Ithaca, N.Y (1985)

[22] Hull R., King R.: *Semantic Database Modeling: Survey, Applications, and Research Issues*, in: ACM Computing Surveys, vol. 19, No. 3, acm Press (1987), 201-260

[23] Hudson S.E.: *Incremental Attribute Evaluation: an Algorithm for Lazy Evaluation in Graphs*, TR No. 87-20, Unviersity of Arizona (1987)

[24] Kastens U.: *Ordered Attribute Grammars*, in: Acta Informatica 13, vol. 13, no. 3, Springer Verlag (1980), 229-256

[25] Kaplan S., Loyall J., Goering S.: *Specifying Concurrent Languages and Systems with Δ-Grammars*, in: [9], 475-489

[26] Kiesel N., Schürr A., Westfechtel B.: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*, in: Information Systems, vol. 20, no. 1, Pergamon Press (1995), 21-51

[27] Lewerentz C.: *Interaktives Entwerfen großer Programmsysteme, Konzepte und Werkzeuge*, Dissertation, RWTH Aachen, Informatik Fachberichte 194, Springer Verlag (1988)

[28] Lipps P., Möncke U., Wilhelm R.: *OPTRAN - A Language/System for the Specification of Program Transformations, System Overview and Experiences*, LNCS 371, Springer Verlag (1989), 52-65

[29] McIntyre D.W., Glinert E.P.: *Visual Tools for Generating Iconic Programming Environments*, in: [42], 162-168

[30] Nagl M.: *Graphgrammatiken*, Vieweg Verlag (1979)

[31] Nagl: *Set Theoretic Approaches to Graph Grammars*, in: [11], S. 41-54

[32] Reps T.: *Generating Language Based Environments*, Dissertation, MIT Press (1984)

[33] Rekers J., Schürr A.: *A Parsing Algorithm for Context-sensitive Graph Grammars (long version)*. Technical Report 95-05, Leiden University, 1995, available by ftp from ftp.wi.leidenuniv.nl, file /pub/CS/TechnicalReports/1995/tr95-05.ps.gz.

[34] Reps T., Teitelbaum T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer Verlag

[35] Snelting G., Bahlke R.: *The PSG System: From Formal Language Definitions to Interactive Programming Environments*, ACM Transactions on Programming Languages and Systems, vol. 8, No. 4, acm Press (1986), 547-576

[36] Schürr A.: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*, Dissertation, Wiesbaden: Deutscher Universitätsverlag (1991)

[37] A. Schürr: *Logic Based Programmed Structure Rewriting Systems*, erscheint in: Rozenberg (ed.): Fundamenta Informaticae, Special Issue on Graph Transformation Systems, Vol. XXVII, No. 1/2 (double issue), Amsterdam: IOS Press (1996)

[38] Schürr A., Winter A., Zündorf A.: *Graph Grammar Engineering with PROGRES*, appears in: Proc. 5th European Software Engineering Conf. (ESEC), Barcelona, Spain, Sept. 1995, LNCS, Berlin: Sprinber Verlag (1995)

[39] Schütte A.: *Spezifikation und Generierung von Übersetzern für Graph-Sprachen durch attributierte Graph-Grammatiken*, Dissertation, EWH Koblenz, EXpress Edition (1987)

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

[41] Proc. of the 1989 IEEE Workshop on Visual Languages, IEEE Computer Society Press (1989)

[42] Proc. of the 1992 IEEE Workshop on Visual Languages, IEEE Computer Society Press (1992)

[43] Proc. of the 1993 IEEE Symp. on Visual Languages, IEEE Computer Society Press (1993)

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

[45] Zündorf A.: *Graph Pattern Matching in PROGRES*, erscheint in: Proc. 5th Int. Workshop on Graph-Grammars and Their Application to Computer Science, Williamsburg, USA, Nov. 1994, LNCS, Springer Verlag (1996)

[46] Zündorf A.:*PROgrammierte GRaphErsetzungsSysteme (Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung)*, Dissertation RWTH Aachen, appears in: Deutscher Universitätsverlag (1995)