Using Programmed Graph Rewriting for the Formal Specification of a Configuration Management System*

Bernhard Westfechtel

Lehrstuhl für Informatik III, RWTH Aachen, Ahornstr. 55, D-52074 Aachen,
bernhard@i3.informatik.rwth-aachen.de

Abstract. Due to increasing complexity of hardware and software systems, configuration management has been receiving more and more attention in nearly all engineering domains (e.g. electrical, mechanical, and software engineering). This observation has driven us to develop a configuration management model (called CoMa) for managing systems of engineering design documents. The CoMa model integrates composition hierarchies, dependencies, and versions into a coherent framework based on a sparse set of essential configuration management concepts. In order to give a clear and comprehensible specification, the CoMa model is defined in a high-level, multi-paradigm specification language (PROGRES) which combines concepts from various disciplines (database systems, knowledge-based systems, graph rewriting systems, programming languages).

1 Introduction

Due to increasing complexity of hardware and software systems, *configuration management* [2] has been receiving more and more attention in nearly all engineering domains (e.g. electrical, mechanical, and software engineering). Configuration management has been defined as the discipline of controlling the evolution of complex systems [17]. In particular, configuration management is concerned with managing system components, their versions, and their interrelations. In this way, configuration management aids in maintaining system consistency.

In this paper, we present a *configuration management model* (called *CoMa* [4]) for managing systems of engineering design documents. The CoMa model integrates composition hierarchies, dependencies, and versions into a coherent framework based on a sparse set of essential configuration management concepts. The model is generic because it is based on principles common to diverse application domains. On the other hand, it can be adapted to a specific scenario (e.g. in a software engineering scenario, documents such as requirements, software architectures or module implementations have to be managed).

* This work was partially supported by Deutsche Forschungsgemeinschaft under the project title SUKITS ('software and communication structures in technical systems').

The CoMa model is defined in a high-level, multi-paradigm specification language (*PROGRES* [15]) which combines concepts from various disciplines (database systems, knowledge-based systems, graph rewriting systems, programming languages). A PROGRES specification is useful in many respects. First, it precisely defines object and relation types by means of a database schema. Second, it describes complex operations on a high level of abstraction. Third, the specification serves as starting point for developing an efficient implementation.

This paper primarily demonstrates an application of the PROGRES specification language; the CoMa model is explained only briefly (see [21] for a more comprehensive description). Section 2 gives a concise informal survey of the CoMa model. Sections 3-5 are devoted to its formal specification. Section 6 describes its implementation, and section 7 compares it to other work. Section 8 concludes the paper.

2 Survey of the Model

The CoMa model represents the following structures which we consider essential for configuration management: The (recursive) *composition hierarchy* describes which components a (sub-)system consists of. Leaf and non-leaf nodes of the hierarchy are called *documents* and *document groups*, respectively. Components of a document group are related by various kinds of *dependencies*. Both documents and document groups evolve into multipe *versions*.

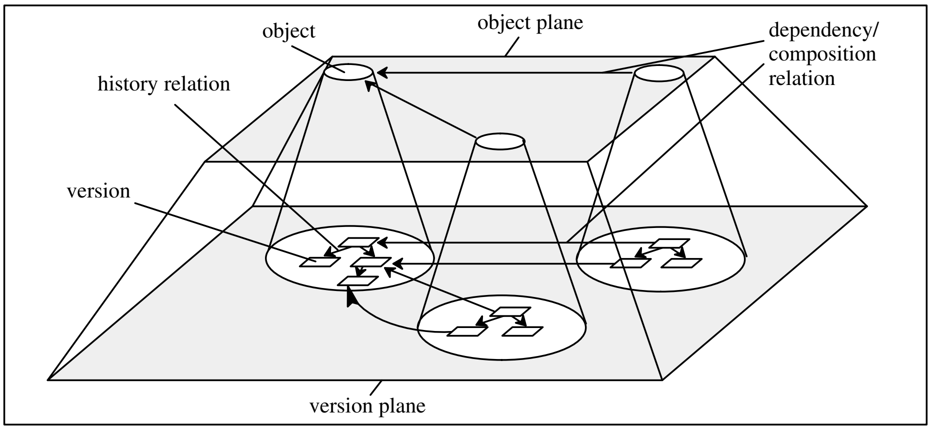


Fig. 1. Object and Version Plane

The CoMa model distinguishes between an *object plane* and a *version plane* (fig. 1). An object represents a set of versions. The version plane refines the object plane: each object is refined into its versions, and each relation between two objects is refined into relations between corresponding versions. Both objects and versions are connected by (hierarchical) composition and (non-hierarchical) dependency relations. Furthermore, history relations between versions of one object represent its evolution.

A database for configuration management is represented by a *CoMa graph* consisting of three kinds of subgraphs which are illustrated in fig. 2. The examples refer to a small program system written in Modula-2. Modula-2 distinguishes between three types of compilation units: program, definition, and implementation modules (denoted by extensions `.p`, `.d`, and `.i`, respectively). Each system contains exactly one program module acting as main program. A module interface is specified in a definition module and realized in the corresponding implementation module. A definition module exports resources (e.g. types or procedures) which may be imported by modules of any type.

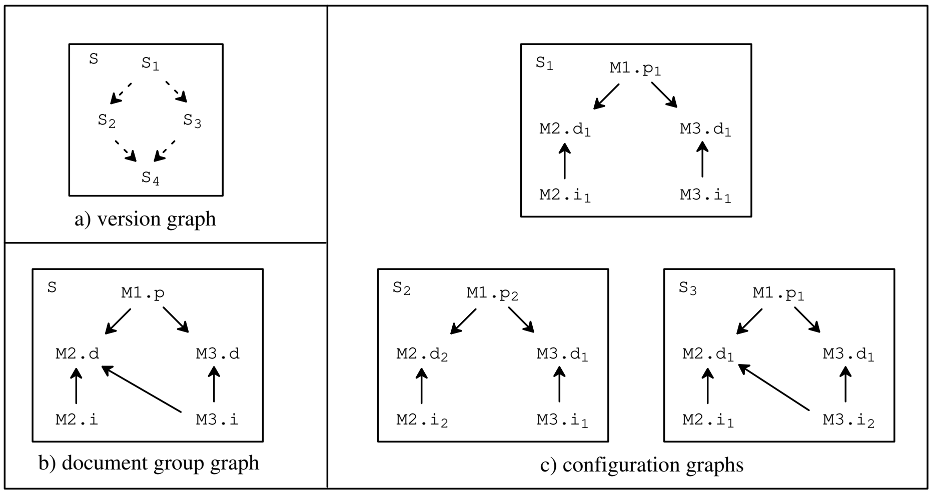


Fig. 2. Subgraphs of CoMa graphs

A *version graph* (fig. 2a) - whose root belongs to the object plane and whose contents is situated in the version plane - represents the evolution history of one object. A history relation from `v1` to `v2` indicates that `v2` is a successor of `v1`. In general, the evolution history is not required to be linear. Rather, it may contain branches (e.g. a bug fix may have to be applied to an old version having already been delivered to a customer) and merges which combine changes performed on different branches.

With respect to the composition hierarchy, versions are classified into revisions and configurations. A *revision* acts as leaf of the composition hierarchy. The internal structure of a revision (e.g. the declarations and statements a module consists of) is not represented within the CoMa graph; rather, a revision is considered an atomic unit (coarse-grained approach). Inner and root nodes of the composition hierarchy are denoted as *configurations*.

A *configuration graph* - which belongs to the version plane - contains version components and their mutual dependencies. Fig. 2c) shows some examples. `S1` (initial configuration of `S`) contains initial revisions of all components. `M1.p1` imports from `M2.d1` and `M3.d1` which are realized by `M2.i1` and `M3.i1`, respectively. The transition to `S2` involves changes to the interface of `M2`, resulting in new revisions of `M2.d`, `M2.i`, and `M1.p`. On the other branch of the evolution history (i.e. in `S3`), the body of `M3` is modified, yielding a new revision `M3.i2` which imports from `M2.d1`.

A *document group graph* - which belongs to the object plane - represents version-independent structural information. In general, the relation between a document group graph and its versions is constrained by the following condition: For each version component (dependency) contained in a configuration graph, a corresponding object component (dependency) must exist in the document group graph. Loosely speaking, a graph monomorphism must exist for each configuration graph into the corresponding document group graph. Fig. 2b) shows a document group graph for `S` which satisfies this condition. Note that for almost all elements of `S`, corresponding elements occur in all configurations of `S` (the only exception consists in the import from `M2.d` into `M3.i`).

3 Schema

After having motivated the CoMa model, we will now turn to its formal specification. We will use the language *PROGRES* [15] which has been developed within the IPSEN project [10]. PROGRES integrates concepts from various disciplines (database systems, knowledge-based systems, graph rewriting systems, programming languages) into one coherent language. A formal semantics for PROGRES based on a logic calculus is described in [14]. In this paper, we will rather be concerned with the application of PROGRES to configuration management problems. In the following sections, we will demonstrate the use of PROGRES by studying a non-trivial, 'real' application ('real' in the sense that a prototypical configuration management system has been built according to the CoMa model).

Let us start with a classical database issue: Each PROGRES specification includes a *schema* which declares types of graph elements. PROGRES is based on attributed graphs consisting of attributed nodes which are connected by binary, directed edges which don't carry attributes. A schema declares node classes and edge types. A *node class* declares attributes, and an *edge type* declares source and target class, as well as its cardinality. Node classes are organized into a (multiple) inheritance hierarchy. A subclass inherits from its superclasses all attributes, and all incoming or outgoing edge types.

Fig. 3 displays a *schema diagram* for the CoMa model. Boxes, dashed and solid arrows denote node classes, inheritance relations, and edge types, respectively. Note that relations which were represented as arrows in fig. 2 are modeled as nodes and adjacent edges (e.g. a history relation is modeled as a `HISTORY` node and `Predecessor` and `Successor` edges). This solution allows for attaching at-

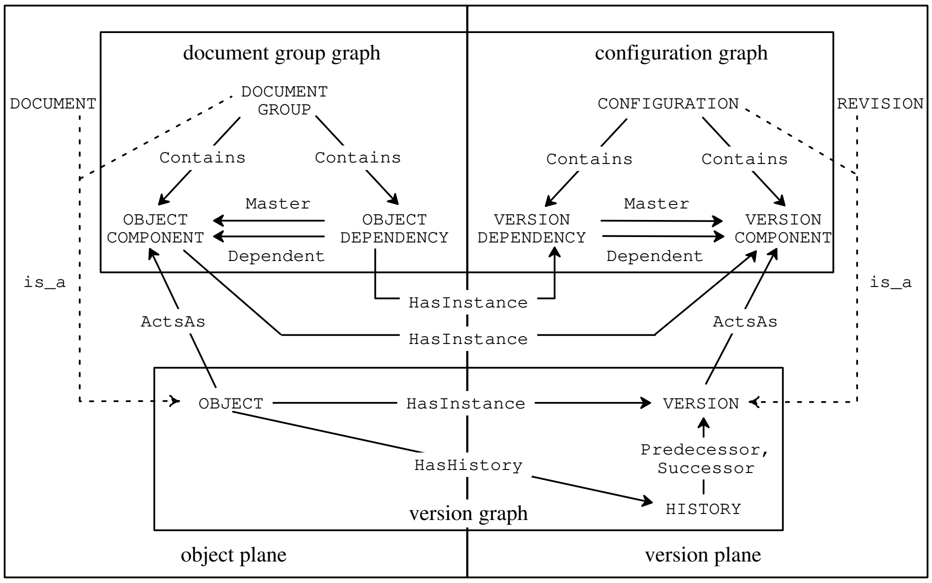


Fig. 3. Schema Diagram

tributes to relations, and establishing relations between relations. In general, relations can be modeled either directly as edges, or as nodes and connecting edges.

Each *subgraph* of the CoMa graph is represented by a root node which is connected to all nodes belonging to this subgraph (e.g. a configuration graph is represented by a `CONFIGURATION` node which is connected to `VERSION_COMPONENT` and `VERSION_DEPENDENCY` nodes by `Contains` edges). The graph model is constructed such that subgraphs are mutually disjoint.

Apart from `HISTORY` nodes, each node of the version plane is connected to the corresponding node of the object plane by an (incoming) `HasInstance` edge. Such a node of the version plane may be regarded as an *instance* of exactly one node of the object plane.

Fig. 4 shows a part of the *textual schema* which refines the overview diagram displayed in fig. 3 by definitions of attributes and cardinalities. Each `OBJECT` node carries an intrinsic `Name` attribute which serves as a unique key. `MaxVersionNo` denotes the number of the next version to be created. Edge type `HasInstance` connects `OBJECT` to `VERSION` nodes. Each object may have any number of versions; conversely, each version is attached to exactly one object (lower and upper bounds of cardinality are enclosed in square brackets). A `VERSION` node carries a number which identifies it uniquely among all versions of one object, a `Stable` attribute indicating whether the version is frozen or may be modified, and two date attributes (`CreationDate` and `LastModificationDate`). Finally, history relations are represented by `HISTORY` nodes and `Predecessor/Successor` edges, and they are connected to `OBJECT` nodes by incoming `HasHistory` edges.

```
section VersionGraphs
   node class OBJECT
      intrinsic
         key Name : string;
         MaxVersionNo : integer := 1;
   end;
   edge type HasInstance : OBJECT [1:1] -> VERSION [0:n];
   node class VERSION
      intrinsic
         VersionNo : integer;
         Stable : boolean := false;
         CreationDate : string := CurrentDate;
         LastModificationDate : string := CurrentDate;
   end;
   edge type HasHistory : OBJECT [1:1] -> HISTORY [0:n];
   node class HISTORY end;
   edge type Predecessor : HISTORY [0:n] -> VERSION [1:1];
   edge type Successor : HISTORY [0:n] -> VERSION [1:1];
end;
```

Fig. 4. Textual Schema for Version Graphs

The schema is not powerful enough to express all kinds of consistency constraints. Fig. 5 summarizes some important *structural constraints* which have (at least partially) already been mentioned in passing. These structural constraints have to be preserved by all operations on the CoMa database.

```
1. Each configuration may contain at most one version of a given object
   (version consistency).
2. Versions of one object have to be numbered in a unique way.
3. A version which has a successor must be stable.
4. All components of a stable configuration must be stable, as well.
5. Cycles in history, dependency, and composition relations are not allowed.
6. Each version component (dependency) has to be mapped "monomorphically" to
   the corresponding object component (dependency).
```

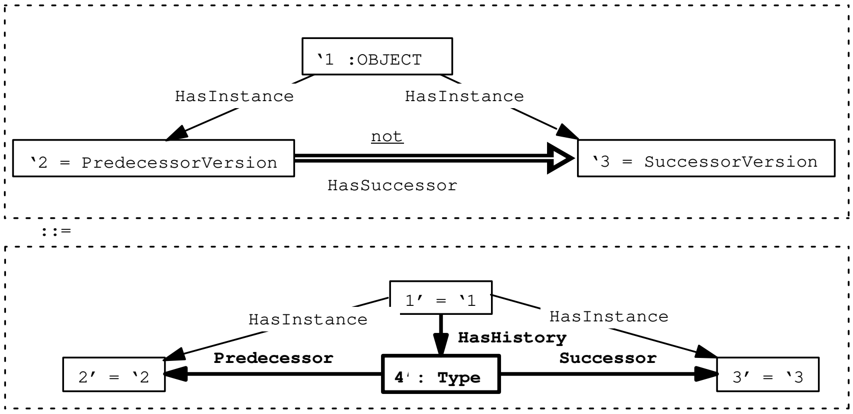
Fig. 5. Consistency Constraints

4 Operations

While many data manipulation languages rely on rather low-level operations such as creating/deleting single entities/relationships or modifying single attributes, PROGRES provides *graph rewrite rules* for specifying complex graph transformations in a declarative and graphical way. In PROGRES, it is possible to specify operations formally and precisely on a high level of abstraction; in many other approaches, you either have to content yourself with some informal, imprecise comments, or you have to deal with lengthy, low-level programs. All graph transformations specified in PROGRES are checked for consistency with the schema at specification time. Thus, PROGRES integrates the database world with the world of graph rewriting systems.

A graph rewrite rule (also called production) consists of the following parts: The *header* is composed of an identifier and a list of formal parameters. The *left-*

```
production CreateHistory
   ( PredecessorVersion, SuccessorVersion : VERSION ; Type : type in HISTORY ;
     out NewHistory : HISTORY )
=
```




```
   condition `2.Stable; `2.VersionNo < `3.VersionNo;
   return NewHistory := 4';
end;
path HasSuccessor : VERSION [0:n] -> VERSION [0:n] =
   (<-Predecessor- & -Successor->)+
end;
```

Fig. 6. Graph Rewrite Rule for Creating a History Relation

hand side describes the subgraph to be replaced. The *right-hand side* specifies the subgraph to be inserted. The *condition part* lists conditions on attributes of nodes belonging to the left-hand side. The *transfer part* assigns values to attributes of nodes belonging to the right-hand side. Finally, result parameters receive values in the *return part*.

The graph rewrite rule `CreateHistory` (fig. 6) receives two *node-valued parameters* (`PredecessorVersion` and `SuccessorVersion`) identifying the versions to be connected by a history relation. Furthermore, it is supplied with a *type parameter* indicating the actual type of history relation to be created. Finally, it returns the new history node as an <u>`out`</u> parameter.

The *left-hand side* contains two `VERSION` nodes which are associated to the same `OBJECT` node by `HasInstance` edges (i.e. both versions must belong to the same object). Nodes `‘2` and `‘3` of the left-hand side are identified with parameters `PredecessorVersion` and `SuccessorVersion`, respectively. Thus, the left-hand side need not be searched globally. The double arrow between node `‘2` and node `‘3` denotes a path condition. The keyword <u>`not`</u> means that the rule is applicable only if there is no path of the specified structure.

The *path declaration* is given below the rule declaration. `<-.-` and `-.->` indicate traversal of an edge in negative/positive direction, respectively; `&` and `(.)+` denote concatenation and transitive closure, respectively. Thus, the <u>`not`</u> condition excludes duplicates of (sequences of) history relations. In general, path conditions are a very powerful and flexible way to specify complex graph patterns.

In the *condition part*, two constraints are checked which were listed in fig. 5: The predecessor version must be stable, and the successor must carry a greater number than the predecessor in order to prevent cycles (constraints 3 and 5 in fig. 5, respectively).

`CreateHistory` is a protective rule, i.e. all nodes and edges of the left-hand side are not affected by its application. Nodes which are replaced identically carry labels of the form `i'='j`. The effect of applying `CreateHistory` consists in creating a new `HISTORY` node (the identifier of which is returned as result parameter) and connecting it to predecessor, successor, and object node, respectively. These insertions are emphasized in bold face on the *right-hand side*.

```
transaction DeleteVersionAndReorganizeHistory
   ( Version : VERSION ; Type : type in HISTORY )
=
     not ( Version is with -ActsAs-> )
   &
   for all PredecessorVersion : VERSION := elem ( Version.HasPredecessor );
           SuccessorVersion : VERSION := elem ( Version.HasSuccessor )
   do
      use NewHistory : HISTORY
      do
         CreateHistory
            ( PredecessorVersion, SuccessorVersion, Type, out NewHistory )
      end
   end
   & DeleteVersion ( Version )
end;
```

Fig. 7. Transaction for a Complex Operation

Although graph rewrite rules may be used to specify rather complex graph transformations, we are convinced that the rule-based specification paradigm alone suffers from severe limitations. PROGRES exceeds the rule-based paradigm by providing *control structures* for the composition of graph rewrite rules [22]. These control structures are similar to those found in procedural programming languages; however, they are designed such that they take atomicity and non-determinism of graph rewrite rules into account.

Fig. 7 shows a *transaction* which makes use of the graph rewrite rule presented in fig. 6. The sample transaction deletes a version and reorganizes the evolution history by connecting all predecessors to all successors. On the top level, its body consists of a *sequence* of statements which are separated by the operator `&`. The first statement asserts that there is no applied occurrence of the version to be deleted. If this assertion is violated, the sequence fails and leaves the host graph unaffected. Note that each control structure preserves *atomicity* of graph rewrite rules, i.e. in case of failure its execution does not affect the host graph. The next statement consists of a *loop* iterating over all predecessors and successors of the current version (the operator `elem` is used to iterate through all elements of a set). Each pair is connected by a history relation (the `use` statement introduces a local variable). Finally, the current version is deleted.

It is beyond the scope of this paper to give a comprehensive description of CoMa operations. Typical examples of *primitive operations* are: create/delete an object; change the name of an object; create/delete/copy a version; create/delete a history relation; ... Based on these primitives, we have also defined *complex operations* which are more convenient to use (e.g. freeze configuration recursively, including all transitive components).

5 Adaptations

So far, the CoMa specification has been independent of a specific application domain. The domain-independent part of the specification is called *generic model*. In the following, we will discuss how the generic model is adapted to a specific domain. The result of such an adaptation is denoted as *concrete model*. As running example, we will use configuration management for Modula-2 programs (see section 2).

The PROGRES type system supports a clear separation between generic model and concrete model. PROGRES has a *stratified type system* which distinguishes between node classes, node types (instances of classes), and nodes (instances of types). Node classes and types are used to specify generic and concrete model, respectively. Nodes are actual instances manipulated at runtime. Due to the stratified type system, types are first order objects which may be supplied as typed parameters, and may be stored as typed values of node attributes.

In order to adapt the generic model, concrete types of documents, document groups, dependencies, etc. have to be defined. Furthermore, operations have to be adapted such that they enforce consistency constraints imposed by the concrete model. For example, in the Modula-2 scenario dependencies from definition modules (dependent) to program modules (master) are prohibited. To achieve this, generic operations are extended such that they access scenario-specific type information. To this end, the schema is enriched by defining *meta attributes* some of which are assigned node types as values. Since meta attributes may only be assigned (i.e. initialized) in node class or node type declarations, their values are type- rather than instance-specific. On the level of the generic model, meta attributes are declared, but not initialized; operations access these attributes. On the level of the concrete model, meaningful values are assigned to meta attributes in node type declarations. In this way, CoMa operations are adapted to a concrete scenario by merely extending the schema and leaving the 'code' of operations unchanged.

To illustrate the approach sketched above, let us describe adaptation of `CreateVersionDependency` to the *Modula-2 scenario*. Fig. 11 displays an excerpt of the specification of this scenario. The composition of the document group type `Program` is given by an ER-like diagram whose boxes and arrows denote component and dependency types, respectively. To both kinds of elements, cardinalities are attached (cardinalities attached to component types actually refer to the composition relations between document groups and their components). Note that the ER diagram applies to both object and version plane.

Let us describe now how such an *ER diagram* is *transformed* into a *PROGRES schema*. We will confine our discussion to version dependencies. In order to perform scenario-specific type checking, meta attributes are attached to nodes of class `VERSION_DEPENDENCY` (fig. 8). `MasterType` and `DependentType` are type-valued attributes (keyword `type in`) which denote the type of master and dependent component, respectively. Boolean attributes `MasterAtMostOnce` and `DependentAtMostOnce` represent upper bounds of cardinality; they are assigned `true` if a given component may play the master or dependent role at most once, respectively. Lower bounds may be defined analogously.

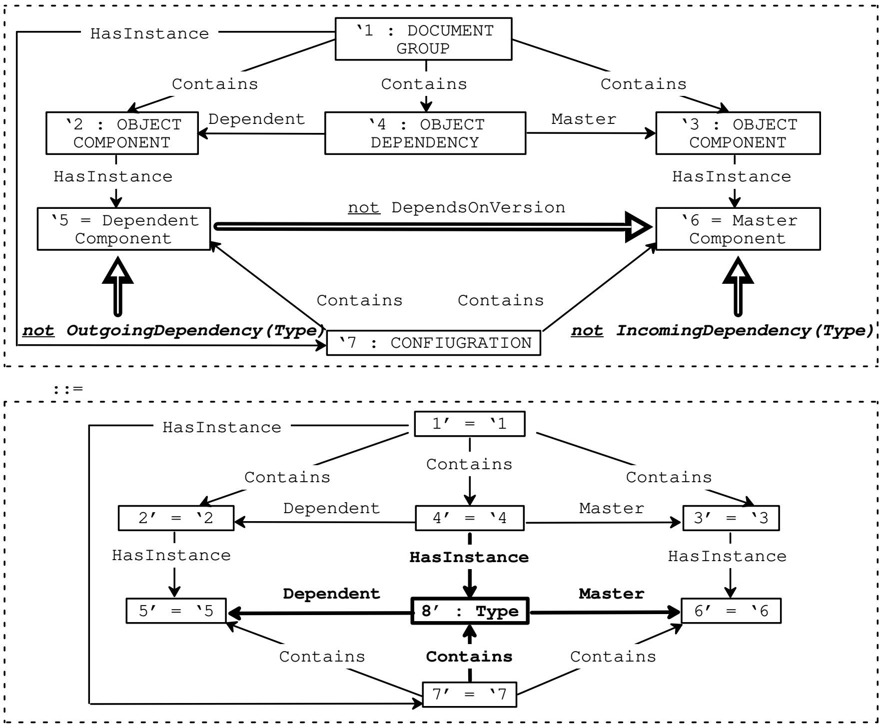
```
node class VERSION_DEPENDENCY
   meta
      MasterType, DependentType : type in VERSION_COMPONENT;
      MasterAtMostOnce, DependentAtMostOnce : boolean;
end;
...
node type ProgModRevisionComponent : VERSION_COMPONENT
   ...
end;
node type DefModRevisionComponent : VERSION_COMPONENT
   ...
end;
...
node type ProgDefImportDependency : VERSION_DEPENDENCY
   redef meta
      MasterType := DefModRevisionComponent;
      DependentType := ProgModRevisionComponent;
      MasterAtMostOnce := true;
      DependentAtMostOnce := false;
end;
```

Fig. 8. Meta Attributes for Specifying Scenario-Specific Constraints

Fig. 8 also shows how these *meta attributes* are *redefined* in case of import dependencies between program and definition modules. `DependentType` and `MasterType` are defined such that components representing program and definition modules may act as dependents and masters, respectively. `DependentAtMostOnce` is assigned `false` because a program module may import from multiple definition modules; conversely, each definition module may act as master at most once because each configuration contains at most one program module component.

Fig. 9 presents the graph rewrite rule `CreateVersionDependency` which receives master and dependent component, and the dependency type as parameters. The rule has to check a lot of constraints enforced by the generic model: master and dependent must belong to the same configuration, the configuration must not be stable, a corresponding object dependency must exist, and there must not yet exist any dependency between master and dependent. As in fig. 6, changes performed by the rule are emphasized in bold face. Furthermore, all elements of the rule concerning *checks of scenario-specific constraints* are printed in bold face and italics. The dependency must be legal with respect to both master and dependent type (see condition part which accesses values of meta attributes associated to dependency type `Type`), and no cardinality overflow must occur (see restrictions applying to nodes `‘5` and `‘6` of the left-hand side). Note that `OutgoingDependency` (`IncomingDependency`) is a *restriction* (not specified in the figure), i.e. a unary relation which is fulfilled if a component already participates in a dependency of a given type with upper bound `1`.

```
production CreateVersionDependency
   ( MasterComponent, DependentComponent : VERSION_COMPONENT ;
     Type : type in VERSION_DEPENDENCY ; out NewDependency : VERSION_DEPENDENCY)
=
```




```
   condition  ‘5.type = Type.DependentType;
              ‘6.type = Type.MasterType;
              not ‘7.Stable;
   transfer 7’.LastModificationDate := CurrentDate;
   return NewDependency := 8’;
end;
```

Fig. 9. Graph Rewrite Rule for Creating a Version Dependency

To conclude this section, let us go into another feature of PROGRES which has a nice application in configuration management, namely maintenance of derived information. A significant portion of configuration management research is devoted to *system building*. In large software systems, it is a rather complex task to build (the executable of) a system correctly, i.e. to trigger compile and link steps in the right order, with correct options, and with minimal effort.

In the sequel, we will sketch how these tasks are supported by means of *derived attributes*. In contrast to intrinsic attributes which are assigned values explicitly, derived attributes are calculated from other attributes attached to the same node or to nodes in the neighborhood. Note that an analogous distinction applies to relations (edges and paths, respectively). Neighbor nodes need not belong to the 1-context; rather, they need only be connected via some path of arbitrary length. The PROGRES runtime system evaluates derived attributes in a lazy fashion, i.e. values are calculated on demand only.

To maintain *compiled code* in the Modula-2 scenario, derived attributes are used in the following way (fig. 10): To each module revision, its source code is attached as an intrinsic `Contents` attribute of type `File`. Object code attributes are attached to all component nodes contained in program configurations (class `PROGRAM_COMPONENT`). Note that `ObjectCode` is declared as an optional attribute (cardinality enclosed in square brackets) because its evaluation will not always succeed.

```
node class REVISION is a VERSION
   intrinsic
      Contents : File;
end;
...
node class PROGRAM_COMPONENT is a VERSION_COMPONENT
   intrinsic
      ObjectCode : File [0:1];
end;
...
node type ProgModRevisionComponent : PROGRAM_COMPONENT
   redef derived
      ...
      ObjectCode =
         [MastersCompiled(self) and def (SourceCode(self))
          ?
          CompileProgMod
          ( SourceCode(self), self.Imports.ObjectCode )
         | nil ];
end;
...
transaction Make
   ( Component : PROGRAM_COMPONENT ; out CompiledComponent : File )
=
   CompiledComponent := Component.ObjectCode
end;
```

Fig. 10. Using Derived Attributes to Specify Compilations

For program modules (node type `ProgModRevisionComponent`), a conditional expression (denoted by `[.|.]`) is given as *evaluation rule*. Since its second alternative evaluates to <u>`nil`</u>, it will yield a defined value only if the first alternative is selected. This alternative is a guarded expression (denoted by `.?.`) which is selected when the guard evaluates to <u>`true`</u>. The guard states that the source of the current component must exist, and that all components on which it depends must have been compiled successfully. In this case, the function `CompileProgMod` is called with two parameters, namely the source code and the set of object codes of all imported components. Within the body of this function, the Modula-2 compiler is called. The function returns nil if compilation fails, and the compiled code otherwise.

After all, it is an easy task to simulate the functionality of the well-known *Make tool* [3]. A call to the function `Make` triggers all necessary compilations in the correct order with minimal effort and delivers the requested object code, if possible. Linking may be handled in an analogous way (attach attribute `Executable` to program module components, define attribute evaluation rules, and provide a function `MakeExecutable`).

6 Implementation

Since PROGRES specifications are executable, the CoMa specification itself may be regarded as a rapid prototype. The CoMa specification covers about 30 pages; it includes all primitive operations and a subset of complex operations offered by the CoMa system (see below). The specification has been developed with the help of the *PROGRES development environment* [11] consisting of editor, analyzer, browser, compiler, and interpreter tools. The PROGRES environment is available as free software.

Starting from the CoMa specification, a configuration management system has been developed within the *SUKITS project* [4] which is dedicated to a posteriori integration of heterogeneous CIM application systems (CAD systems, CAD systems, NC systems, etc.). The *CoMa system* developed in the SUKITS project consists of a schema editor for adapting the system to a specific scenario (a screen dump of which is shown in fig. 11), tools for editing, browsing, and analyzing configuration management data on the instance level, interfaces to CIM application systems to be integrated, and an OSI-based communication system gluing all components together in a heterogeneous environment (multiple types of machines, operating systems, and data management systems). The implementation of the CoMa system comprises more than 60,000 loc written in Modula-2 and C; about 50 % of the code are dedicated to the implementation of operations on the CoMa database. The implementation of the CoMa system was made considerably easier by reuse of components developed within the IPSEN project [10], which is concerned with the construction of integrated software development environments.

7 Relation to Other Work

To the best of our knowledge, the work presented in this paper (and previous work performed by the author [19, 20]) is *unique* in *applying* a specification language based on *programmed graph rewriting* to *configuration management* of engineering design documents. We hope to have convinced the reader that the PROGRES specification language is well-suited to describe and manipulate complex graph structures occurring in this application domain.

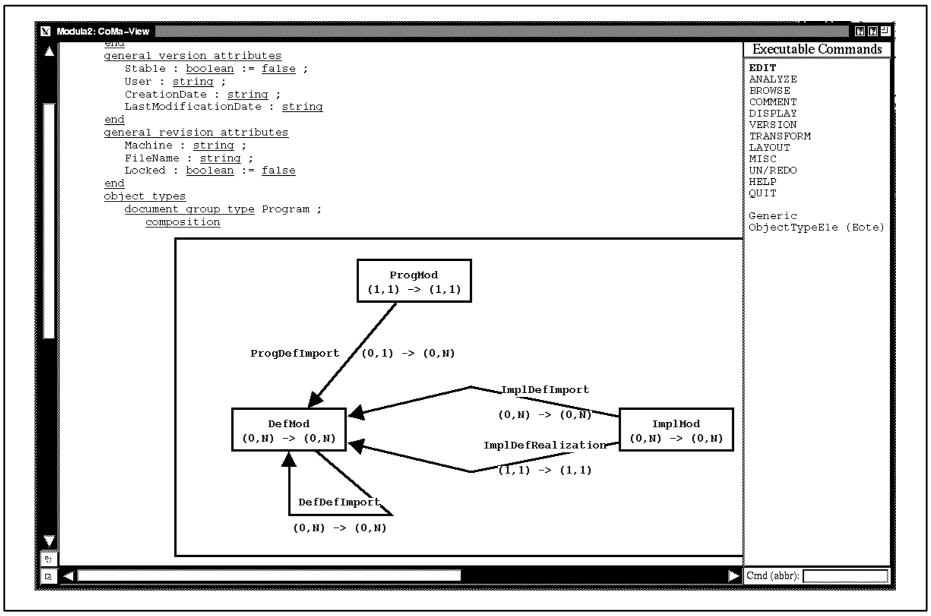


Fig. 11. Screen Dump from the CoMa Schema Editor

Comparing the *CoMa model* and the *functionality* of the CoMa system to other configuration management systems, we observe the following: Traditional configuration management tools such as Make [3], which supports consistent production of derived objects, or SCCS [13] and RCS [16], which both efficiently store revisions of text files, have been very successful. However, each of these tools solves just one configuration management problem, and integration is possible only to a limited extent. Furthermore, they are file-based and do not rely on a formal data model.

More recently, the advantages of using *databases* have been acknowledged [7], and configuration management tools and systems have been developed either on top or as built-in components of database systems. This approach opens the door for configuration management which benefits from the semantic expressiveness of their underlying data models. DAMOKLES [6] and PCTE [12], which both support versions of complex objects based on some extended ER data model, may be quoted as examples. More comprehensive approaches to configuration management, which integrate versioning, composition, and dependencies into a coherent framework, have been realized e.g. in the Nelsis CAD framework [18] and the DEC Cohesion environment [1].

To conclude this section, let us briefly compare PROGRES to other approaches based on *graph rewriting*. Research conducted in this area has mainly been driven by theoretical computer scientists who put a strong emphasis on developing a sound theory. Only recently, this situation has begun to change (e.g. recent extensions to the categorical approach which increase expressive power of graph rewrite rules at the expense of loosing some theoretical properties [8, 9]). On the other hand, the design of PROGRES has strongly been driven by application domains such as software engineering or database systems from its very beginning. By designing a specification language and implementing an integrated development environment, we have moved away from the 'paper and pencil mode' of applying graph rewriting systems. In these respects, the intentions of PROGRES are similar to those followed by Göttler [5] which has designed and implemented tools for editing and executing programmed graph rewriting systems. However, his approach is different in many respects (e.g. type system, control structures, attributes).

8 Conclusion

We have presented a configuration management model (called CoMa) for managing systems of engineering design documents. The CoMa model integrates composition hierarchies, dependencies, and versions into a coherent framework based on a sparse set of essential configuration management concepts. In order to give a clear and comprehensible formal specification, the CoMa model has been defined in the PROGRES language. With the exception of non-determinism and backtracking (which play only a minor role in the CoMa specification), we have exploited more or less the full range of constructs provided by the PROGRES specification language (schema definition, derived attributes and relations, stratified type system, graph rewrite rules, control structures, transactions). We are convinced that PROGRES is superior to other approaches relying on rather low-level data manipulation primitives; furthermore, we believe that we actually need the expressiveness of a multi-paradigm specification language. Our experiences we have gained in configuration management have strongly confirmed these attitudes.

References

1. *A Tool Integration Standard*, ANSI Draft, Digital Equipment Corporation (1990)
2. Feiler, P. *Configuration Management Models in Commercial Environments*, Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1991)
3. Feldman, S.I. *Make - A Program for Maintaining Computer Programs*, Software - Practice and Experience, vol. 9, 255-265 (1979)
4. Große-Wienker, R., Hermanns, O., Menzenbach, D., Pollack, A., Repetzki, S., Schwartz, J., Sonnenschein, K., Westfechtel, B. *Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme*, Aachener Informatik-Berichte 93-11, RWTH Aachen (1993)
5. Göttler, H. *Graphgrammatiken in der Softwaretechnik - Theorie und Anwendungen*, Informatik Fachberichte 178, Springer Verlag (1987)
6. Gotthard, W. *Datenbanksysteme für Software-Produktionsumgebungen*, Informatik Fachberichte 193, Springer Verlag (1988)

7. Katz, R.H. *Toward a Unified Framework for Version Modeling in Engineering Databases*, ACM Computing Surveys, vol. 22-4, 375-408 (December 1990)
8. Löwe, M. *Extended Algebraic Graph Transformation*, Dissertation, Technical University Berlin (1991)
9. Löwe, M. *Single-Pushout Transformation of Attributed Graphs: A Link between Graph Grammars and Abstract Data Types*, Proc. SEMAGRAPH Symposium 1991, John Wiley & Sons, 359-379 (1991)
10. Nagl, M. *Characterization of the IPSEN Project*, in: Madhavji, Schäfer, Weber (Eds.): Proc. of the 1st Int. Conf. on Systems Development Environments & Factories 1989, Pitman Press, 141-150 (1990)
11. Nagl M., Schürr A. *A Specification Environment for Graph Grammars*, in: Ehrig, Kreowski, Rozenberg (Eds.): Graph Grammars and Their Application to Computer Science, Proc. of the 4th Int. Workshop 1990, LNCS 532, Springer Verlag, 599-609 (1991)
12. Oquendo, F. et al. *Version Management in the PACT Integrated Software Engineering Environment*, Proc. of the 2nd European Software Engineering Conference, 222-242 (1989)
13. Rochkind, M.J. *The Source Code Control System*, IEEE Transactions on Software Engineering, vol. 1-4, 364-370 (December 1975)
14. Schürr, A. *Operationale Spezifikation mit programmierten Graphersetzungen: Formale Definitionen, Anwendungen und Werkzeuge*, Deutscher Universitäts Verlag (1991)
15. Schürr, A. *PROGRES: A VHL-Language Based on Graph Grammars*, in: Ehrig, Kreowski, Rozenberg (Eds.): Graph Grammars and Their Application to Computer Science, Proc. of the 4th Int. Workshop 1990, LNCS 532, Springer Verlag, 641-659 (1991)
16. Tichy, W.F. *RCS - A System for Version Control*, Software : Practice and Experience, vol. 15-7, 637-654 (July 1985)
17. Tichy, W.F. *Tools for Software Configuration Management*, Proc. International Workshop on Software Version and Configuration Control, Teubner Verlag, Stuttgart, 1-20 (1988)
18. van der Wolf, P., Bingley, P., Dewilde, P. *On the Architecture of a CAD Framework: The Nelsis Approach*, Proc. 1st European Design Automation Conference, IEEE Computer Society Press, 29-33 (1990)
19. Westfechtel, B. *Revision Control in an Integrated Software Development Environment*, Proc. of the 2nd International Workshop on Software Configuration Management, ACM SIGSOFT Software Engineering Notes, vol. 14-7, 96-105 (November 1989)
20. Westfechtel, B. *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*, Informatik Fachberichte 280, Springer Verlag (1991)
21. Westfechtel, B. *A Graph-Based System for Managing Configurations of Engineering Design Documents*, Aachener Informatik-Berichte AIB 94-15, Technical University of Aachen (1994)
22. Zündorf, A., Schürr, A. *Nondeterministic Control Structures for Graph Rewriting Systems*, Proc. 17th Int. Workshop on Graph-Theoretic Concepts in Computer Science WG '91, LNCS 570, Springer Verlag, 48-62 (1991)

This article was processed using the LaTeX macro package with LLNCS style