A Graph–Based System for Managing Configurations of Engineering Design Documents

Bernhard Westfechtel
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, D–52074 Aachen
e–mail: 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 domain–independent and adaptable 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). Finally, we also present an implementation which conforms to the formal specification and provides graphical, structure–oriented tools offering a bunch of sophisticated commands and operating in a heterogeneous environment.

1. Introduction

Due to increasing complexity of hardware and software systems, configuration management [21] 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 [52]. When many engineers cooperate in development or maintenance of a system consisting of many components evolving into many versions, chaos seems to emerge almost inevitably. Application of configuration management methods and tools aims at reducing such chaos and providing engineers with a safe and organized working environment.

The CoMa system [59] assists in managing versioned systems of engineering design documents. CoMa provides a lot of valuable features:

- CoMa is domain–independent, i.e. it may be applied to different domains of engineering design. In particular, we have applied CoMa to both software engineering and mechanical engineering.
- On the other hand, CoMa is adaptable to a certain application domain. Configuration management operations offered by the CoMa system are tailored by an ER–like schema which defines object and relation types specific to a certain application domain.

† This work was partially supported by the German Research Council (DFG) under the project title SUKITS (software and communication structures in technical systems).
• Since CoMa has been designed for a broad range of applications, its underlying model is based on a sparse set of reusable configuration management concepts, namely composition hierarchies, versioning, and dependencies.

• Rather than supporting these concepts in isolation, CoMa integrates them into a coherent framework. In particular, CoMa provides uniform versioning of both documents and document groups (revisions and configurations, respectively), nested configurations, and control of consistency between interdependent, versioned components of configurations.

• Since the CoMa system has to handle complex configurations, its design and implementation has been a challenging task. Therefore, we have defined the model underlying the CoMa system in a high–level specification language (PROGRES [45, 46, 63]) which combines concepts from various disciplines (database systems, knowledge–based systems, graph rewriting systems, programming languages).

The contributions of this paper are twofold. Firstly, it presents a system which provides practical support for solving complex configuration management problems. Secondly, it illustrates the use of a powerful specification language. Although the first contribution should not be neglected, we have decided to put more emphasis on the second one. Within our research group (IPSEN [39]), we have been designing and implementing integrated, structure–oriented environments (mainly for software engineering applications) for a long time. Driven by our experience, we have been developing a specification language and supporting tools in order to alleviate the development of applications operating on complex data structures. PROGRES is a multi–paradigm specification language which is based on attributed graphs and supports definition of graph schemas, a stratified type system, derived attributes and relations, graph rewrite rules, transactions, control structures, non–determinism, and backtracking. The current paper describes a non–trivial application of PROGRES to configuration management and therefore serves as a case study to evaluate the rich features offered by this language.

Fig. 1. Basic notions and their interrelations

Fig. 1 depicts relations between some important notions used throughout the paper. Starting from informal concepts, the CoMa model is formalized by writing a PROGRES specification. Such a specification has a precise meaning because the semantics of the PROGRES language is defined by means of a logic calculus. Implementation of the CoMa model yields the CoMa system which is written in Modula–2 and C.
The paper is structured according to fig. 1. In section 2, the underlying concepts of configuration management are described in an informal way. Section 3 presents a cutout of the PROGRES specification which defines the CoMa model. Section 4 is devoted to the CoMa system which implements the CoMa model. Section 5 performs a comparison to related work. A short summary concludes the paper (section 6).

2. Informal Description

Configuration management for engineering design documents provides an essential service function for getting work done in an organized way. Engineering design documents describe products from various perspectives which are highly interrelated. Configuration management has to keep track of these dependencies. In this way, engineers are assisted in maintaining consistency between interrelated documents. To this end, documents and their interdependencies are arranged in configurations. Note that management of these configurations must also take evolution into account. During their lifetimes, documents evolve into multiple versions which have to be recorded accurately. Rather than treating versions of individual documents in isolation, configuration management must handle dependencies between versions of different documents, as well. In this way, configuration management provides a global organization of engineering design documents over long periods of time.

In the following, we describe the basic notions and principles underlying the CoMa model. We focus on the way in which engineering design documents are organized in order to support version control, consistency control, and configuration control. The informal description given below will be formalized in the next section.

CoMa manages engineering design documents (briefly denoted as documents in the sequel). For example, documents such as requirements definitions, software architectures, and modules have to be managed in software engineering, while we have to deal with designs, manufacturing plans, and NC programs in mechanical engineering. A document is a logical unit of reasonable size typically manipulated by a single engineer. Since CoMa is domain–independent, it does not rely on any assumptions regarding the internal structure of documents. This approach is called coarse–grained because a document is considered an atomic unit. Physically, a document is typically represented as a file or a complex object in an engineering database.

Logically related documents are collected in document groups. For example, a document group may comprise all documents describing a certain software system. Although all documents of a group might be created by the same engineer, a document group frequently serves as a work context for integrating documents produced by multiple engineers. In general, document groups may be nested so that they form a composition hierarchy of arbitrary depth.

Components of a document group are related by various kinds of dependencies. Such dependencies may either connect components belonging to the same work area (e.g. dependencies between modules of a software system), or they may cross work area boundaries (e.g. dependencies between software architectures and requirements definitions). Accurate management of dependencies is an essential prerequisite for consistency control, i.e. for keeping interdependent components consistent with each other. To this end, tools are needed which operate on the fine–grained level, i.e. on the contents of documents. Since CoMa follows a coarse–grained approach, the CoMa system itself cannot perform consistency control on the fine–grained
level. However, it manages the results of consistency analyses and provides a reusable framework for embedding domain–specific tools operating on the fine–grained level.

![ER–like diagram for Modula–2 programs](image)

**Fig. 2.** ER–like diagram for Modula–2 programs

To illustrate the notions of document groups and dependencies, fig. 2 shows an ER–like diagram for the running example used throughout this paper, namely configuration management for Modula–2 programs. Note that such a diagram is used to adapt CoMa to a certain application domain. The diagram defines types of documents and dependencies which constitute a program written in Modula–2: Document groups of type `Program` contain compilation units (called modules in Modula–2) of various types. Each program contains exactly one `ProgramModule` acting as main program. A module interface is specified in a `DefinitionModule` and realized in the corresponding `ImplementationModule`. A `DefinitionModule` exports resources (e.g. types or procedures) which may be imported by modules of any type.

During their evolution history, both documents and document groups evolve into multiple versions. Versions may be regarded as snapshots recorded at appropriate points in time. The reasons for managing multiple versions of an object (instead of just its current state) are manifold: reuse of versions, maintenance of old versions having already been delivered to customers, storage of back–up versions, or support of change management (e.g. through a diff analysis figuring out what has changed compared to an old version).

CoMa supports uniform versioning of both documents and document groups. Therefore, we subsume both notions under the term *object*. An object is a generic entity which represents the common properties of a set of versions. These properties are expressed either by attributes (e.g. object name) or by relations. Each object has an evolution history represented by a graph of versions interconnected by *history relations*.

A version is a specific instance of a generic object. In addition to the properties common to all versions, each version has specific properties which are expressed by attributes and relations, as well. Versions of documents and document groups are denoted as *revisions* and *configurations*, respectively. Due to uniform versioning, all benefits gained from version control (e.g. reuse, change management) may be exploited on arbitrary levels of the composition hierarchy.
According to the distinction between objects and their versions, the CoMa model distinguishes between an object plane and a version plane (fig. 3). 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. Furthermore, history relations between versions of one object represent its evolution. While the object plane provides an overview by abstracting version-independent structural information, the version plane accurately represents actual revisions and configurations as well as their mutual relations.

Objects, versions and their interrelations are formally represented by graphs. In order to structure the information contained in such graphs, the CoMa model distinguishes between different kinds of interrelated subgraphs, namely version, configuration, and document group graphs, respectively. Each subgraph provides a view on the underlying database displaying the evolution history of an object, the structure of a configuration, and the structure of a document group, respectively.

A version graph consists of versions which are connected by history relations. As an example, fig. 4 displays the version graph of a multi-pass compiler which is used as running example throughout this paper. A history relation from $v_1$ to $v_2$ indicates that $v_2$ was derived from $v_1$ (usually by modifying a copy of $v_1$). In simple cases, versions are arranged in a sequence reflecting the order in which they were created. Concurrent development of multiple versions causes branches in the evolution history. In such a case, a version has multiple successors. Conversely, a version which is constructed by merging branches has multiple predecessors. Finally, a version graph may even be separated (not shown in the example) if multiple branches have
been developed in parallel from the very beginning. To summarize, the CoMa model only enforces the constraint that a version graph must not contain cycles. Variants can be represented by branches from a common ancestor or by separated subgraphs.

A **configuration graph** contains version components and their mutual dependencies. As an example, fig. 5 displays two configurations of the multi-pass compiler. A version component is identified by an object name, a dot, and a version number. In the first configuration, Control plays the role of the main program which sequentially invokes the passes LexAna, SynAna, SemAna, and CodeGen (lexical, syntactic, semantic analysis, and code generation, respectively). The TextFile submitted to the compiler is transformed step by step (→ TokenSeq → ParseTree → GraphCode →) into a BinFile. Both TextFile and BinFile are realized using a general-purpose module File. The structure of the second configuration is slightly different: an optimization pass (Optimize) has been added which operates on the intermediate graph code. New and changed elements are emphasized by bold lines and diagonal stripes, respectively.

A **document group graph** represents version-independent structural information in the object plane. In general, the following constraint must hold between a document group graph and the corresponding configuration graphs in the version plane: For each version component (dependency) contained in a configuration graph, a corresponding object component (dependency) must exist in the document group graph. More precisely, each configuration graph must be mapped by a graph monomorphism into the corresponding document group graph. This constraint guarantees that the version plane actually refines the object plane. Furthermore, an injective mapping excludes that multiple versions of some object are contained in a configuration (version consistency, which is enforced e.g. in Modula–2, Ada and many other programming languages). Fig. 6 displays a document group graph which meets the refinement constraint with respect to the
Fig. 5. Examples of configuration graphs
configuration graphs of fig. 5. Note that optional elements (for the optimization pass) are indicated by dashed lines.

Fig. 6. Example of a document group graph

Document group graphs support management of system families existing in many variants. For example, the multi–pass compiler might vary with respect to the following attributes: target machine, host machine, algorithms for lexical, syntactic, or semantic analysis, presence/absence of an optimization pass, etc. Each member of the family is a specific configuration of the multi–pass compiler. Although the members of system families should have much in common, it is too restrictive to enforce identical structures (graph isomorphisms). Rather, structural variations have to be permitted (an example was given above). Thus, a structural variant is a certain subgraph of the document group graph. Such a variant can be specified as follows: For each document group, a set of variant attributes is defined. To each component and dependency, a variant description is attached which refers to these attributes and controls its visibility. Then, a variant is specified by assigning values to the variant attributes and evaluating the variant descriptions.

Experience shows that structural variants must be managed carefully. If the differences between family members become too large, the document group graph is cluttered up with lots of variant–specific components and dependencies. As a consequence, the document group graph gets unintelligible, and the structures of configuration graphs differ significantly from each other and from the structure of the document group graph. In such a situation, the document group graph is no longer helpful. It is then advisable to split the document group such that the variations in each of the resulting families are reduced significantly. Thus, the
user has to decide upon an appropriate organization, and he has to employ the concepts offered by the CoMa model in a disciplined and reasonable way.

3. Formal Specification

After having sketched the CoMa model informally, we now turn to its formal specification. To this end, we use the language PROGRES [46, 48]. In subsection 3.1, we briefly introduce PROGRES and motivate its application to configuration management. The remaining subsections describe cutouts of a PROGRES specification of the CoMa model. The language constructs are explained as required; further information on the language can be found in [45, 63].

3.1. Applying PROGRES to configuration management

Building tools for engineering design applications is a challenging task. Such tools operate on complicated internal data structures representing e.g. CAD designs, manufacturing plans, SA diagrams, software architectures, or programs. In order to realize commands to be executed on such documents, sophisticated operations on the corresponding internal data structures have to be performed. Before implementing these operations in a programming language such as C, Ada, or Modula–2, a high–level specification is needed from which the implementation may be derived.

Driven by our experience in the construction of integrated, structure–oriented environments (mainly for software engineering applications), we have been developing a specification language which satisfies this need. The PROGRES language integrates concepts from various disciplines (database systems, knowledge–based systems, graph rewriting systems, programming languages) into one coherent language: PROGRES is based on attributed graphs consisting of attributed nodes which are connected by binary, directed edges. A schema defines types of nodes, edges, and attributes. In particular, derived attributes and relations may be defined in a schema. Graph transformations are specified by high–level graph rewrite rules rather than described by low–level operations such as creation or deletion of single nodes or edges. Furthermore, PROGRES supports programming with graph rewrite rules in order to realize even more complex graph transformations. Finally, the language takes non–deterministic choices (e.g. between different occurrences of the left–hand side of a graph rewrite rule) into account.

To sum up, the language features listed above justify to denote PROGRES as a specification language. A specification written in PROGRES lies on a much higher level of abstraction than a program written in a conventional programming language. More specifically, the term specification may be characterized as follows:

- A PROGRES specification is formal because there is a sound denotational definition of the language’s semantics [47]. Due to non–determinism, the meaning of a PROGRES specification is defined as a relation rather than a function. The semantics definition is based on a logic calculus. Graphs, graph schemas, and graph rewrite rules are represented by sets of logical formulas in a uniform way. The application of a graph rewrite rule is defined as a manipulation of the logical formulas representing the host graph (for further details, the reader is referred to [45, 47]).
A PROGRES specification is *model–oriented* [62] because it provides a (graph) model for some abstract datatype. Operations are defined in terms of their effects on the model rather than by axioms as e.g. in the algebraic approach [61]. As a consequence, PROGRES specifications are executable (operational specification) and can be used for rapid prototyping (see also section 4).

<table>
<thead>
<tr>
<th>features of PROGRES</th>
<th>application to configuration management</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph schema</td>
<td>structure of version, configuration, and document group graphs</td>
</tr>
<tr>
<td>stratified type system</td>
<td>separation between generic and specific model</td>
</tr>
<tr>
<td>derived attributes and relations</td>
<td>object code generation</td>
</tr>
<tr>
<td>graph rewrite rules</td>
<td>atomic graph transformations (e.g. create history relation)</td>
</tr>
<tr>
<td>transactions, control structures</td>
<td>complex graph transformations (e.g. delete version and reorganize history)</td>
</tr>
</tbody>
</table>

Fig. 7. Application of PROGRES to configuration management

PROGRES supports verification of specification properties through a rich static semantics. Many errors in a specification are caught by checking its correctness with respect to the language rules for static semantics. In particular, in a correct PROGRES specification all operations are guaranteed to preserve the properties expressed in the schema. Over 300 rules are checked by the analyzer and therefore need not be verified manually. On the other hand, some checks are still beyond the scope of the analyzer and can be performed only at runtime. Due to the complexity of the PROGRES language, complicated structural constraints (e.g. refinement relation between object and version plane) cannot be verified automatically, i.e. there is no automatic theorem prover. However, such constraints can be described in PROGRES in a graphical notation, and constraint–preserving graph rewrite rules can be derived in a mechanical way.

In the following subsections, we formalize the CoMa model by providing a PROGRES specification. Fig. 7 summarizes the main features of PROGRES and their application to configuration management. All topics appearing in this table will be detailed below.

3.2. **Schema**

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 all attributes, and all incoming or outgoing edge types from its superclasses.
Fig. 8 displays a schema diagram for the CoMa model. Boxes, dashed and solid arrows denote node classes, inheritance relations, and edge types, respectively. Relations which were represented as arrows in section 2 are modeled as nodes and adjacent edges (e.g., a history relation is modeled as a HISTORY node and Predecessor/Successor edges). This solution allows for attaching attributes to relations, and establishing relations between relations. In general, relations can be modeled either directly as edges, or indirectly 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. In particular, the model distinguishes between a version (node class VERSION) and an applied occurrence of a version in a configuration (node class VERSION_COMPONENT). This means that the composition hierarchy is expressed via a level of indirection. As a consequence, context–dependent information may be attached to a version component. For example, the compiled code for a module depends on the compilation context defined by a configuration (subsection 3.4).

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. In this way, the refinement relation between object plane and version plane is represented.
Fig. 9 displays a cutout of a schema–compatible graph. The example refers to the multi-pass compiler which was introduced in the last section. The upper half of the figure shows a cutout of a configuration graph for configuration 4 of the Compiler. Among others, the configuration contains version components Optimize_d.1, Optimize_i.2 (definition and implementation module for the optimization pass, respectively), and GraphCode_d.1 (definition module for the intermediate code representation). These components are connected by two dependencies: Optimize_i.2 realizes Optimize_d.1, and it imports GraphCode_d.1. The lower half of the figure illustrates cutouts of three version graphs. In the version graph for Optimize_i, revisions 1 and 2 are connected by a history relation. Elements of version graphs and the configuration graph are linked by ActsAs edges.

Fig. 9. Cutout of a schema–compatible graph

Fig. 10 shows a part of the textual schema which refines the overview diagram displayed in fig. 8 by definitions of attributes and cardinalities. In principle, the complete schema could be represented in one diagram; we have factored out the details in order to keep fig. 8 as clear as possible. 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 cardinalities 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,
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];

path HasSuccessor : VERSION [0:n] -> VERSION [0:n] =
  <-Predecessor- & -Successor->
end;

end;

Fig. 10. Textual schema for version graphs

1. Objects must be named in a unique way.
2. Versions of one object have to be numbered in a unique way.
3. An object must not act more than once as a component in the same document group.
4. History and dependency relations must be local, i.e. source and target must belong to the same subgraph.
5. A version which has a successor must be stable.
6. All components of a stable configuration must be stable, as well.
7. Cycles in history, dependency, and composition relations are not allowed.
8. Each version component (dependency) has to be mapped ‘monomorphically’ to the corresponding object component (dependency).
9. For each pair of versions (components), there must exist at most one connecting history (dependency) relation.

Fig. 11. Consistency constraints
history relations are represented by HISTORY nodes and Predecessor/Successor edges, and they are connected to OBJECT nodes by incoming HasHistory edges.

At the bottom of fig. 10, a simple example of a path declaration is given. A path declaration defines a derived relation in terms of operators for traversing outgoing and incoming edges (\(\rightarrow\) and \(<\rightarrow\), respectively), concatenation (\&), transitive closure \((.)^+\), etc. The path HasSuccessor connects a version to all successors by traversing incoming Predecessor and outgoing Successor edges.

```plaintext
node class NODE;
  derived Constraint : boolean;
end;

node class HISTORY is a NODE;
  redef derived Constraint = self is HistoryRestriction;
end;

restriction HistoryRestriction : HISTORY = '3 in
  condition '2.Stable; '2.VersionNo < '5.VersionNo;
end;

restriction HistoryRestriction : HISTORY =
  exist v1 : VERSION := self.–Predecessor–>,
    v2 : VERSION := self.–Successor–>,
    o : OBJECT := self.–HasHistory–>
    (v1 in o.–HasInstance–>) and (v2 in o.–HasInstance–>) and
    (v1.Stable) and (v1.VersionNo < v2.VersionNo) and
    (not (exist h : HISTORY := o.–HasHistory–>
      (h.–Predecessor–> = v1) and (h.–Successor–> = v2) and (h \neq self))
end
end;
```

**Fig. 12.** Specification of consistency constraints

So far, the graph schema is not powerful enough to express all kinds of consistency constraints. Further structural constraints are listed in fig. 11. They are formalized in the following way (fig. 12): In the declaration of class NODE (the root of the class hierarchy), a derived attribute Constraint is introduced. Defining expressions for this attribute are given in subclasses of NODE. In this way, each node class defines the constraints in which it is involved. All nodes with Constraint=false are considered erroneous.
As an example, fig. 12 specifies the constraints which have to hold for history relations. The defining expression for the attribute `Constraint` in the class `HISTORY_RELATION` refers to a `restriction`, i.e. a unary relation which may be specified either textually or graphically:

- The graphical restriction states that a node of class `HISTORY_RELATION` (node ‘3’) has to belong to a certain graph pattern. In particular, the restriction checks constraints 4, 5, 7, and 9. Node ‘4’ is used to express a negative application condition, i.e. the restriction is only met if no match for this node can be found.
- An equivalent textual restriction is given below the graphical restriction. This notation may be preferred by users who are familiar with predicate logic. Note that the textual restriction closely corresponds to the meaning which is attached to the graphical restriction by the semantics definition of PROGRES (as we have mentioned in section 3.1, the graph schema is mapped onto a set of logical formulas).

During development of the specification, constraint definitions may be used in two ways. As we are going to demonstrate in the next subsection, graphical restrictions provide valuable assistance in constructing constraint-preserving graph rewrite rules. Furthermore, the `Constraint` attributes may be used to debug a specification.

### 3.3. Operations

PROGRES provides *graph rewrite rules* for specifying graph transformations in a declarative and graphical way. ‘Declarative’ means that the user is not concerned with the algorithm that is executed in order to apply a graph rewrite rule. In particular, the user need not worry about how an instance of the left-hand side is searched in the host graph (once found, replacement with the right-hand side is rather straightforward).

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-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. Left-hand side and right-hand side are mandatory, all other parts are optional.

In the following, we give two examples of graph rewrite rules. The first one describes an insertion, the second one a deletion. The examples presented below do not only illustrate the rich facilities of PROGRES for specifying replacements of subgraphs. Furthermore, they also demonstrate the systematic construction of consistency-preserving rules. In many cases, it is a straightforward task to derive a graph rewrite rule from the relevant constraints stated in the schema.

Let us first consider an insertion rule. Most (but not all) primitive insert operations create a single node and some adjacent edges. For each node class, a graphical restriction (see fig. 12 for an example) defines the constraints which have to hold for its instances. The following heuristic usually yields a consistency-preserving insertion rule:

1. The header is constructed as follows: Supply the type of the node to be created as input parameter and its identifier as output parameter (each node has a unique identifier). Furthermore, supply (the identifier of)
each context node as input parameter provided that it cannot be determined uniquely by navigation.
Finally, add an input parameter for each uninitialized intrinsic attribute.
2. To obtain the left–hand side, remove the node to be created from the restriction graph.
3. To obtain the right–hand side, remove all positive and negative application conditions from the restriction
   graph and take the input type as type of the new node.
4. The condition part is copied from the condition part of the restriction.
5. Assign the values of intrinsic attributes in the transfer part.
6. Assign the identifier of the new node to the output parameter in the return part.

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

= 

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'1 = OBJECT</td>
<td>HasInstance</td>
<td>HasInstance</td>
</tr>
<tr>
<td></td>
<td>HasHistory</td>
<td>Successor</td>
</tr>
<tr>
<td>HasInstance</td>
<td>Predecessor</td>
<td>'4 = HISTORY</td>
</tr>
</tbody>
</table>

'2 = PredecessorVersion

return NewHistory := '4';
end;

Fig. 13. Graph rewrite rule for creating a history relation

The reader may easily verify that application of this heuristic to the graphical restriction of fig. 12 yields
the rewrite rule for creating a history relation which is shown in fig. 13. Inserted nodes and edges are shown in
bold. Note that a node label of the form ‘i=p on the left–hand side indicates that node ‘i is fixed by input
parameter p. Moreover, j’='i on the right–hand side means that node ‘i is replaced identically.

Fig. 14 displays another example of a graph rewrite rule which demonstrates the use of set nodes (dashed
double rectangles). Provided there is no applied occurrence (negative node ‘6), DeleteVersion deletes a
version, all incoming and outgoing history relations, and the contents of the corresponding configuration
graph (if any) in a single rewrite step. To achieve this, set nodes are used which are mapped onto maximal sets
of nodes in the host graph. Note that nodes ‘4 and ‘5 cannot be connected directly to node ‘2 via
Contains edges because this would cause a type error. Therefore, a path is needed whose first element checks whether the source node actually belongs to class CONFIGURATION (the `instance_of` clause in the path expressions beside the double arrows acts as a filter).

The rule presented above serves an example for a deletion rule. In general, a graph rewrite rule for deleting a node has to take its context into account. The following heuristic guides the construction of a graph rewrite rule for deleting a node of class C:

1. Insert the node into the left-hand side.
2. Determine all context nodes which may be affected by a deletion. In contrast to the rule shown above, it is in general not sufficient to take the 1-context into account. Rather, transitively affected nodes must be considered, as well.
3. For each type of context node, decide whether its existence prohibits deletion. If the answer is ‘yes’, add a negative application condition to the left-hand side. If the answer is ‘no’, then insert a single or set node (depending on the cardinality of the set of context nodes) into the left-hand side.
4. Leave the right-hand side empty.

Although graph rewrite rules may be used to specify rather sophisticated 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 [63]. 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. 15 shows a `transaction` which makes use of the graph rewrite rules presented in fig. 13 and 14. 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. Like graph rewrite rules,
compound statements have the atomicity property, i.e. they either succeed completely or fail and leave the host graph unaffected. The next statement consists of a loop iterating over all predecessors and successors of the current version. Each pair is connected by a history relation (the use statement introduces a local variable). Finally, the current version is deleted.

So far, we have only given examples for specifying graph transformations. In addition, PROGRES provides language constructs for read operations. Atomic and complex read operations are specified by tests and queries, respectively. A test corresponds to a graph rewrite rule which describes an identical replacement. A test succeeds if an instance of a graph pattern occurs in the host graph. out parameters are used to return information about nodes and attributes. A query is specified in the same way as a transaction, but it may only contain actions which leave the host graph unaffected.

Fig. 16 shows a test which returns the components and dependencies of a structural variant of a document group (see end of section 2). Set-valued parameters return the qualifying nodes. A variant description \( VD \) is supplied as input parameter. The restriction for node \('2\) requires that the Variants attribute of an object component match the variant description \( VD \). In addition, the restriction for node \('3\) demands that source and target of the dependency be contained in the structural variant, as well. In this way, referential integrity of the structural variant is guaranteed.

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 defined complex operations which are more convenient to use. We have decided to add these operations, leaving all primitive operations still accessible. Let us list some typical examples of complex operations: delete version and reorganize history (fig. 15); derive version, i.e. copy version and create history relation; purge history, i.e. delete all unused versions of an object; update configuration, i.e. bind all version components to the most recent version; freeze configuration recursively (including all transitive components); ...
3.4. 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 discuss how the generic model is adapted to a specific domain. The result of such an adaptation is denoted as concrete model.

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 meta 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, meta model 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.

This approach is illustrated by the graph rewrite rule for creating a version dependency which is given in fig. 18 (to be explained later on). This rule checks the values of meta attributes which are defined in fig. 17. These values are assigned according to the ER diagram of fig. 2.

In fig. 17, meta attributes 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 cardinalities; they are assigned `true` if a given component may play the master or dependent role at most once, respectively.

```plaintext
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. 17. Meta attributes for specifying scenario–specific constraints

Fig. 17 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. 18 presents the graph rewrite rule `CreateVersionDependency` which receives the master and the dependent component, as well as 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. 13, changes performed by the rule are emphasized in
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;

restriction OutgoingDependency ( Type : type in VERSION_DEPENDENCY ) : VERSION_COMPONENT =
'1 in

collection Dependent
condition Type.DependentAtMostOnce;
end;

Fig. 18. Graph rewrite rule for creating a version dependency
bold face. Furthermore, all elements of the rule concerning checks of scenario–specific constraints are printed in 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). OutgoingDependency (IncomingDependency) is a restriction, i.e. a unary relation, which is fulfilled if a component already participates in a dependency of a given type with upper bound 1 (the cardinality is checked within the condition part of the restriction).

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 software 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 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. 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.

```
node class REVISION is a VERSION
  intrinsic
    SourceCode : File;
end;
...
node class PROGRAM_COMPONENT is a VERSION_COMPONENT
  intrinsic
    ObjectCode : File [0:1];
end;
...
node type ImplModComponent : PROGRAM_COMPONENT
  redef derived
    ObjectCode =
      {self.MastersCompiled and
       self.InterfaceCompiled and
       def (self.BoundTo.SourceCode)
         CompileImplMod
         (self.BoundTo.SourceCode,
          self.Realizes.ObjectCode,
          self.Import.ObjectCode)
       | nil }
    ;
end;
...
transaction Make
  ( Component : PROGRAM_COMPONENT ; out CompiledComponent : File )
  =
    CompiledComponent := Component.ObjectCode
end;
```

Fig. 19. Using derived attributes to specify compilations
To maintain compiled code, derived attributes are used in the following way: The outputs of compilers and linkers are stored in derived attributes. Compilation and link steps are triggered by read accesses to these attributes. System building is controlled by the attribute evaluator which operates in an incremental and lazy fashion. As in Make [22], only those modules whose object codes are out of date are recompiled in the correct order. The attribute evaluation rules are set up according to the compilation dependencies. During attribute evaluation, compilers and linkers are activated by calling external functions (i.e. functions which are not written in PROGRES, but may be used in a PROGRES specification through an import interface).

In the Modula–2 scenario, derived attributes are defined in the following way (fig. 19): To each module revision, its SourceCode is attached as an intrinsic attribute of type File. Object code attributes are attached to all component nodes contained in program configurations (class PROGRAM_COMPONENT). ObjectCode is declared as an optional attribute (cardinality enclosed in square brackets) because its evaluation will not always succeed. ObjectCode is a context–dependent attribute which in general varies across multiple configurations. Therefore, it cannot be attached to revisions; rather it has to be attached to component nodes which represent applied occurrences of revisions in configurations.

For implementation modules (node type ImplModComponent), a conditional expression (denoted by \[. | .\]) is given as evaluation rule. Since its second alternative evaluates to \texttt{nil}, it will yield a defined value only if the first alternative is selected. This alternative is a guarded expression (denoted by \texttt{.?}) which is selected when the guard evaluates to \texttt{true}. The guard states that the source of the current component must exist, and that the corresponding definition module and all imported components must have been compiled successfully. In this case, the function CompileImplMod is called with three parameters, namely the source code, the object code of the corresponding definition module, and the set of object codes of all imported components. Within the body of this external function (not shown in the figure), the Modula–2 compiler is called. The function returns \texttt{nil} if compilation fails, and the compiled code otherwise.

![Fig. 20. Example for the use of derived attributes](image-url)
After all, it is an easy task to simulate the functionality of the well–known Make tool. A call to the function `Make` (bottom of fig. 19) triggers all necessary compilations in the correct order with minimal effort – due to lazy evaluation of derived attributes – 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`).

Fig. 20 presents an example which illustrates maintenance of compiled code by means of derived attributes (see also fig. 9, subsection 3.2). Dashed lines represent attribute dependencies. The `ObjectCode` attached to component `Optimize_i.2` (node 4) depends on the `SourceCode` of the revision (node 10) to which the component is bound (the path `BoundTo` traverses an `ActsAs` edge in opposite direction). Furthermore, it depends on the `ObjectCode` attribute of the corresponding definition module component `Optimize_d.1` (node 2) and (among others) on the `ObjectCode` attribute of the imported component `GraphCode_d.1` (node 6). If the transaction `Make` is applied to `Optimize_i.2`, read access to its `ObjectCode` attribute will trigger all necessary recompilations in the correct order.

### 4. Implementation

The specification presented in the last section was developed with the help of the PROGRES environment [48] which offers tightly integrated tools for editing, analyzing, browsing, compiling, and interpreting specifications. Since PROGRES specifications are executable, a working prototype can be constructed by compiling a PROGRES specification into a target language such as Modula–2 or C. Only recently, a generator has been developed which adds a graphical user interface to the compiled code and offers commands for applying tests, graph rewrite rules, queries, and transactions. In this way, a working implementation may completely be generated from a specification. Thus, programming is eliminated, and there is also no need to prove the correctness of the implementation with respect to the specification.

For historical reasons, the CoMa system was not implemented in this way. When the initial version of the CoMa system was implemented in 1993, neither the PROGRES compiler nor the user interface generator were available. Instead, we used components developed in the IPSEN project [39]. IPSEN is concerned with the design and implementation of integrated, structure–oriented environments; it addresses software development and maintenance as its main application domain. Starting from the PROGRES specification, an implementation was developed which fits into the IPSEN framework. In particular, this required transforming CoMa graphs, as described in the PROGRES specification, into a normalized form amenable to the IPSEN environment (abstract syntax graphs).

The CoMa system comprises about 30,000 lines of code written in Modula–2 and C. It consists of a schema editor, editors for document groups, configurations, and version graphs, and various analysis tools. The schema editor is used to adapt tools operating on instances to a certain application domain. Arbitrary schema modifications (not only extensions) may be performed provided they do not result in inconsistencies. Instance–level tools respect both generic and specific consistency constraints. Each command is checked immediately for constraint violations; commands which would introduce inconsistencies are rejected. Many high–level commands are offered to the user (see also end of subsection 3.3).
All tools provide a hybrid user interface which combines textual and graphical representations. The graphs which are employed internally are presented in a user–friendly way. Currently, the PROGRES user interface generator does not support user–friendly representations; rather, it presents graphs as they are. Due to this shortcoming, we have not yet considered reimplementing the CoMa system with the help of the PROGRES generator. A 1:1 presentation of CoMa graphs would be illegible (see e.g. fig. 9).

Fig. 21. User interface of the CoMa system

Fig. 21 illustrates the user interface of the CoMa system. The window at the bottom displays a configuration of the multi–pass compiler (our running example) which is being extended with an optimization pass. An analysis has been invoked which searches the configuration for unbound components. The component Optimize is highlighted because it has not yet been bound to a specific revision (so far, no implementation of the optimization pass has been produced). The top window displays the corresponding document group. An analysis has been invoked which returns optional components, i.e. components which do not occur in all configurations. Definition and implementation module for the optimization pass as well as incoming and outgoing dependencies are marked. Note that the user need not update the document group explicitly while
So far, the only application domain having been discussed in this paper has been software engineering. In particular, we have focused on managing configurations of Modula–2 programs in order to present a running example which is easy to understand. However, we have also applied CoMa to a different application domain, namely mechanical engineering. To this end, the SUKITS project [20] was launched in fall 1991. SUKITS is a joint project of computer scientists and mechanical engineers which aims at developing an integrating infrastructure for heterogeneous CIM application systems. Within this project, management of engineering design documents has been investigated in two scenarios, namely development of single metal parts and of plastic parts produced by injection moulding, respectively. Both scenarios are rather complex; the corresponding schemas both comprise about 15 document types and 30 dependency types. Documents to be managed include e.g. CAD designs for different purposes (e.g. geometries of the single part to be produced, of the raw part to start with, of the positions of the work piece, etc.), manufacturing plans (CAP), NC programs (CAM), material requirements definitions, results of FEM simulations (CAE), etc. The CoMa tools play a central role in the SUKITS system; for further details, the reader is referred to [60].

5. Relation to Other Work

In the following, we relate our approach to other work in the field of configuration management. The comparison is carried out from two perspectives and refers to the functionality as well as to the specification of the CoMa system (subsections 5.1 and 5.2, respectively).

5.1. Functionality

Research in configuration management for engineering design documents has been carried out in different domains [11]. The most prominent application domains which have been investigated so far are software engineering and CAD. For a survey of these domains, the reader is referred to [21, 52] and [30], respectively. Furthermore, we also take a short look at configuration management in office automation.

For a long time, software configuration management (SCM) has been dominated by tools and environments which have been developed on top of (or even integrated into) file–based systems. Early efforts in this field succeeded in providing isolated tools for solving specific SCM problems. Make [22], which supports consistent production of derived objects, and SCCS [44] or RCS [51], which both efficiently store revisions of text files, may be quoted as classical examples. Subsequently, tool–kit environments such as DSEE [33, 37], shape [38], or ClearCase [34] were built which integrate and extend the functionality of first–generation SCM tools. Design and implementation of these tools and environments were mainly driven by the goal to support programming–in–the–large (and to a certain extent programming–in–the–many) on top of file–based operating systems. However, software configuration management systems which rely on database systems have also been developed more recently. Most of them are based on EER (Extended Entity–Relationship) models (e.g. PCTE [41], Adele [19], and DAMOKLES [16]) or object–oriented models (e.g. the ATIS class hierarchy [4] which forms the basis of Digital’s COHESION environment [56] and Atherton’s Software Backplane [3]).
Although many similarities exist between software engineering and CAD approaches, research in configuration management for CAD applications has been founded mainly on databases rather than on file systems from its very beginning. Systems like Orion [10], the Version Server [8], Cadlab [6], or the Nelsis CAD framework [53] manage versions of complex objects which are maintained in different representations (e.g. functional, layout, and logical representations in VLSI design). Among the approaches developed for CAD applications, OVM (object–version model [29]) is most closely related to CoMa. OVM is realized on top of the PRIMA database system [25] which is based on an EER model providing built–in support for complex objects. Like CoMa, OVM distinguishes between an object and a version plane. Unlike CoMa, configurations are neither versioned, nor may they be nested. To sum up, OVM strongly resembles a predecessor of the CoMa model which was published for the first time in 1989 [57, 58].

Office automation is another domain in which configuration management problems have been addressed. This domain differs from CAD and software engineering as informal documents (e.g. scanned paper documents) have to be managed rather than documents with a well–defined syntax and semantics. Therefore, consistency control (as for source and derived objects in software configuration management and multiple views in configuration management for CAD applications) does not play an important role. Document organization is supported by hypertext systems such as e.g. Neptune [12], which provides hypertext links as well as simple version control facilities. Furthermore, there are document processing systems such as e.g. TEXPROS [55] which arrange documents into folders and describe them by attributes. The strength of TEXPROS lie in its automatic document categorization facility, its flexible, content–based folder organization, and its powerful query facilities. However, version control and consistency control are not explicitly addressed.

The survey presented above shows that configuration management systems in different domains are based on rather similar concepts (although a common and coherent conceptual framework has not yet been developed [30]). In the following, we summarize the design decisions which we consider most essential for the development of the CoMa system. Furthermore, we characterize the position of CoMa within the spectrum of other configuration management approaches:

- CoMa uses graphs to represent the complex structures encountered in configuration management. In particular, this distinguishes CoMa from the main stream of file–based software configuration management systems.
- CoMa combines composition hierarchies, versioning, and dependencies in an integrated framework. In contrast, many other systems only support a subset of these dimensions (e.g., dependencies in Make, versioning and – in a restricted way – configurations in RCS).
- In CoMa, versioning is applied uniformly on all levels of the composition hierarchies. In this respect, CoMa is more general than e.g. RCS or OVM which do not support versioning of configurations.
- In CoMa, the hierarchy is represented via a level of indirection: A configuration consists of version components which refer to nested versions. As a consequence, context–dependent information (attributes and relations) may be attached to a version component (e.g. compiled code). In particular, this modeling approach distinguishes CoMa from other systems based on EER models (e.g. OVM, PCTE, and DAMOKLES) which directly connect configurations to nested versions.
- Dependencies between components of a configuration play a crucial role in the CoMa system. Firstly, they document the structure of a configuration; secondly, they are used for consistency control. In many other approaches, including PCTE, ATIS, and DAMOKLES, there is no built–in support for dependencies (only general relations), or dependencies are only represented implicitly (e.g. in the Version server
equivalence objects subsume different representations of one logical object, but the dependencies between representations are not modeled explicitly).

- The CoMa model distinguishes between an object plane and a version plane. The version plane refines the object plane. System families are supported by maintaining version-independent structural information (document group graphs) in the object plane. To the best of our knowledge, OVM is the only other system supporting object and version plane (see the comparison above).

5.2. Specification

As far as we know, the work presented in this paper (and previous work performed by the author [57, 58]) is unique in applying a specification language based on programmed graph rewriting to configuration management of engineering design documents. We see many benefits in this approach, including a precise and formal description not only of graph structures, but also of operations. Most other configuration management systems lack an appropriate formal foundation. In particular, this applies to file-based software configuration management systems. But even configuration management systems which rely on databases are usually based on a formal specification only to a limited extent (database schemas, no formal specification of operations).

Since PROGRES is a multi-paradigm language, it can be compared to a variety of other languages in different domains. In the following, we relate PROGRES to specification languages, database programming languages, and languages based on graph rewriting (the most relevant domains with respect to this paper).

Many formal specification languages have been developed for different purposes. Since PROGRES has been designed for the design of sequential systems, we exclude specification languages for concurrent systems from our comparison. According to [62], we may distinguish between property-oriented and model-oriented approaches. Unlike e.g. Larch [61] which belongs to the class of algebraic specification languages, PROGRES is a model-oriented language. It differs from other languages such as VDM [28] and Z [50], which also support model-oriented specifications, with respect to the underlying data model. In VDM and Z, operations are specified on data structures such as sets, lists, sequences, or mappings. In contrast, PROGRES is solely based on graphs. VDM and Z do not provide built-in support for graphs, but graphs can be defined within these languages. Thus, VDM and Z can handle more general structures than PROGRES. On the other hand, PROGRES provides specific support for the definition and manipulation of graphs.

PROGRES has been designed to support the development of engineering design environments (in particular for software engineering). Therefore, we may view PROGRES as a database programming language for engineering design applications. Different data models have been applied in this domain:

- Abstract syntax trees have been used in various syntax-aided software development environments (or generators for such environments) such as e.g. Gandalf [24] and CPSG [42]. Software configuration management serves as an excellent example for demonstrating the advantages of using a more general data model than trees (see also the discussion in section 4).
- Relational database systems [31] have been applied successfully to business applications for a long time. They have a well-founded theoretical background and provide powerful facilities for formulating queries as well as defining schemas and views. However, the relational data model does not match nicely the
complex, heterogeneous structures encountered in engineering design applications. This observation has launched a great bunch of research activities in database systems for this application domain [5].

- Database systems such as DAMOKLES [16] and PCTE [54], which have been developed specifically for engineering design applications, are based on EER models (which ultimately have their roots in the ER model proposed by Chen [9]). EER models are better suited for the representation of complex, heterogeneous objects than the relational data model (which is adequate for large amounts of homogeneous data). In particular, they provide built-in support for complex objects and often even for versions. Compared to PROGRES, the main weakness of most EER-based systems refers to the specification of updates. In PROGRES, graph rewrite rules are used to this end, while in EER-based systems updates have to be broken down to elementary operations for deleting/inserting/changing single entities/relationships/attributes. Note that this statement also applies to database systems which are based on an object-oriented data model (e.g. O₂ [15], Objectstore [32], and GemStone [7]).

To conclude this section, let us 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 [17]. 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 [35, 36]). 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.

There are a few other systems which aim at practical applications of graph rewriting. WSDW [13, 14] is a software development environment workbench whose tool specification language supports definition and manipulation of graph structures with an underlying spanning tree (unlike PROGRES which relies on general graphs). EdGen [2] supports the generation of graph editors and is built upon a rather restricted approach to graph rewriting (no attributes, no derived information, no programming). GOOD [1] is a database system which is based on attributed graphs. Unlike PROGRES, graph rewrite rules in GOOD do not support paths, set nodes, and embedding transformations (the latter of which were not used in this paper). Finally, PAGG [23] is based on programmed, attributed graph grammars. However, PAGG only provides poor support for typing. While derived attributes are declared in the graph schema in PROGRES, PAGG attaches attribute evaluation rules to graph rewrite rules (implying that the derivation tree must be maintained in addition to the host graph in order to enable evaluation of derived attributes). Finally, non-determinism is not handled adequately in PAGG (no backtracking).

6. Conclusion

We have presented a configuration management model (called CoMa) for managing systems of engineering design documents. The model distinguishes between a version plane and an object plane. A CoMa database is formally represented as a graph containing three types of subgraphs: version graphs capturing the evolution histories of objects, configuration graphs containing version components and their mutual dependencies, and document group graphs providing version-independent information on the structure of systems or subsystems.
In order to give a clear and comprehensible formal specification, the CoMa model has been defined in the PROGRES language. PROGRES combines concepts from various disciplines (database systems, knowledge–based systems, graph rewriting systems, programming languages) into a multi–paradigm, high–level specification 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.

Acknowledgements

I am indebted to Marita Breuer, Ursula Cordts, Gregor Jöris, and Cornelia Weigmann for implementing the CoMa system. Michael Becher, Peter Heimann, Arno Jansen, and Joachim Schwartz extended the CoMa system in order to use it in the SUKITS environment. Furthermore, I appreciate very much the contributions of Andy Schürr, the designer of PROGRES, who suggested many useful improvements to the specification of the CoMa model. Finally, I acknowledge the valuable comments provided by the reviewers.

References


