Pergamon

*Information Systems* Vol. 20, No. 1, pp. 21-51, 1995

0306-4379(95)00002-X
0306-4379/95 $9.50 + 0.00

# GRAS, A GRAPH-ORIENTED (SOFTWARE) ENGINEERING DATABASE SYSTEM†

NORBERT KIESEL, ANDY SCHUERR and BERNHARD WESTFECHTEL

RWTH Aachen, Informatik III, Ahornstr. 55, D-52056 Aachen, Germany

*(Received 18 March 1994; in final revised form 2 November 1994)*

**Abstract** — Modern software systems for application areas such as software engineering, CAD, or office automation are usually highly interactive and deal with rather complex object structures. For the realization of these systems a nonstandard database system is needed which is able to efficiently handle different types of coarse- and fine-grained objects (like documents and paragraphs), hierarchical and non-hierarchical relations between objects (like composition-links and cross-references), and finally attributes of rather different size (like chapter numbers and bitmaps). Furthermore, this database system should support incremental computation of derived data, undo/redo of data modifications, error recovery from system crashes, and version control mechanisms. In this paper, we describe the underlying data model and the functionality of GRAS, a database system which has been designed according to the requirements mentioned above. Furthermore, we motivate our central design decisions concerning its realization, and report on experiences and applications.

*Key words:* attributed graphs, graph rewriting systems, distribution, version control, derived data, software engineering environments

## 1. INTRODUCTION

Building integrated, interactive, and incrementally working tools and environments for application areas like software engineering, CAD, or office automation has been a great challenge. Among other factors such as presentation, control, and process integration, data integration is a key issue. In order to ensure that all information in an environment is managed as a consistent whole, it should be modeled and realized in a uniform way. To this end, a database system is needed which takes the specific requirements of these environments into account [9, 74].

In this paper, we present such a database system which is called **GRAS** (for **GRA**ph **S**torage, [10, 48, 49, 55]). GRAS has been developed within the IPSEN project [61] which is concerned with integrated, structure-oriented software engineering environments. Since the IPSEN project was launched in the mid 80's, GRAS has been extensively used for the implementation of integrated, structure-oriented environments (not only in IPSEN, but also in other research projects). Thus, a lot of experience has been gained, resulting in continuous extensions and improvements of GRAS in response to the requirements of its applications.

In order to cope with the complexity of information managed in an integrated, structure-oriented environment, its underlying database system has to be based upon an appropriate data model. This data model should allow us to represent complex object structures in a natural way. While the relational model has proved to be adequate for traditional business applications, it is not well-suited for building applications such as software engineering environments, hypertext systems, CAD systems, etc. [9, 74]. On the other hand, it seems quite natural to model objects as nodes (with attributes representing their properties) and relations between them as edges. Therefore, we have selected **attributed graphs** as the data model underlying the GRAS system.

The kernel of GRAS is tuned to efficient management of medium-sized, complex data of dynamically varying size and structure. In this way, it accounts for typical characteristics of applications such as structure-oriented editors, analyzers, and execution tools. The functionality of the kernel is extended by multiple layers implemented on top of the kernel. These layers provide for event handling, change management, incremental computation of derived data, and client/server distribution.

†Recommended by N. Gehani

The presentation is structured as follows: Section 2 describes the data model underlying the GRAS system. Sections 3 and 4 present the architecture of the GRAS system. Section 3 describes the kernel, while section 4 focuses on enhancing layers. Section 5 discusses experiences with and applications of GRAS. Section 6 compares the GRAS system with related work, and section 7 concludes the paper.

## 2. DATA MODEL

In order to cope with the complexity of information managed in an integrated, structure-oriented environment, a formal specification language called **PROGRES** [70, 71, 73] has been developed within the IPSEN project. A precise definition of this language including syntax, static and dynamic semantics (the latter one being defined by a calculus based on first-order predicate logic) is given in [71, 72]. PROGRES is based on attributed graphs and provides constructs for defining graph schemes (data definition) and complex graph transformations (data manipulation). Its name is an acronym for **PRO**grammed **G**raph **RE**writing **S**ystems: Graph transformations are specified graphically by means of graph rewrite rules which describe the replacement of a left-hand side subgraph with a right-hand side subgraph. The attribute "programmed" indicates that the application of these graph rewrite rules is controlled by deterministic and nondeterministic control structures.

GRAS and PROGRES are related in the following way: GRAS provides basic operations on graphs such as creation/deletion of nodes and edges, manipulation and incremental computation of attributes, etc. These operations are checked against a graph scheme which is defined by means of the data definition part of PROGRES. Complex graph transformations specified in PROGRES are mapped onto basic operations provided by GRAS (which is the task of a compiler/interpreter). Thus, GRAS serves as kernel of a more comprehensive database development environment for PROGRES. However, GRAS can—and has actually been—used as an independent component outside the PROGRES environment (see section 5). The PROGRES environment should be viewed as one—albeit important—application of GRAS which is a reusable component in its own right.

In this section, we will focus on the definition of graph schemes which control the operations performed by the GRAS database system. Therefore, we have to omit the most important part of PROGRES, its visual data manipulation rules. The presentation will be informal, based on a simple example (for a comprehensive and formal description, see [71]).

Graph schemes describe the components of attributed graphs: Objects are represented by means of typed nodes which may carry attributes. Binary, directed relations between objects are modeled by edges which don't carry attributes. Each edge has two distinct ends which are denoted as source and sink, respectively. Edges are bidirectional, i.e. they may be traversed in both directions (from source to sink and vice versa). Furthermore, referential integrity is guaranteed, i.e. when a node is deleted, all adjacent edges are deleted, as well. Attributes are either intrinsic (the value is assigned explicitly) or derived. In the latter case, the value is automatically calculated from values of other attributes which belong to the same node or to neighbor nodes. Analogously, we distinguish between intrinsic relations (edges) and derived relations the latter of which are specified by path expressions.

To provide an illustrative example, fig. 1 shows a small cut-out of a famous textbook. The text is structured hierarchically with paragraphs forming the leaves of the tree. Each section has a name (e.g. *"The phases of a compiler"*) and is numbered in a hierarchical fashion (e.g. number *1.3*). The composition tree is augmented with cross-references which may be created between arbitrary components (e.g. from the first paragraph of section *1.3* to section *1.5*).

Fig. 2 displays the corresponding attributed graph. Each **node** is represented by a box whose header contains a unique identifier and a type (e.g. *1 : Book*) and whose body lists for each attribute its name and its value. An attribute is either single-valued (e.g. *Name = "Compilers: ..."*) or set-valued (e.g. *Authors* = {*"Aho"*, *"Sethi"*, ...}). The *Number* attribute attached to *Section* nodes is calculated automatically by means of a derivation rule. This rule uses the *Position* attribute (which is in turn a derived attribute) and will be explained below. All remaining attributes are intrinsic, i.e. their values are assigned explicitly. The *Contents* attribute attached

Aho, Sethi, Ullman: Compilers: Principles, Techniques, and Tools

1 Introduction to Compiling
The principles and techniques of compiler writing are so pervasive that the ideas found in this book will be used many times in the career of a computer scientist. Compiler writing spans programming languages, machine architectures, language theory, algorithms, ....
...
1.3 The phases of a compiler
Conceputally, a compiler operates in phases, each of which transforms the source program from one representation to another. ... In practice, some of the phases may be grouped together, as mentioned in Section 1.5, ...
1.4 Cousins of the compiler
1.5 The grouping of phases
...
2 A Simple One-Pass Compiler
3 Lexical Analysis
...

Fig. 1: Structured text with cross-references

to *Paragraph* nodes (e.g. node 5) serves as an example of long attributes. Finally, **edges** are represented by arrows with labels indicating their types. *Contains* edges span the composition tree, *Precedes* edges define a linear order on sibling components, and *RefersTo* edges represent cross-references (e.g. from node 9 to node 8).

Fig. 3 shows a graphical representation of the **graph scheme** underlying the hypertext graph of fig. 2. Currently, a graph scheme is specified in the PROGRES language by means of a textual notation which we will introduce below as required.

As demonstrated in fig. 2, each node is an instance of a node type each of which is represented as a bold-faced box at the bottom of fig. 3. A node type determines which attributes its instances carry, and in which relations they participate. Node types are used for type checking of operations to which nodes are passed as parameters. Since it is sometimes useful to supply not only nodes, but also node types as typed parameters for graph operations, PROGRES has a **stratified type system**: Each node is an instance of a node type which, in turn, is an instance of a node class (i.e. node classes are types of node types). For example, the node type *Book* is an instance of node class *DOCUMENT* (other instances of this class might be e.g. *TechnicalReport*, *Paper*, etc.). The stratified type system avoids the theoretical pitfalls of reflexive type systems with the 'type is the type of all types including itself' assumption (cf. [58]). Furthermore, it distinguishes clearly between abstract and concrete types (node classes and types, respectively) which in many object-oriented data models are mixed together in one inheritance hierarchy.

**Node classes** model sets of nodes with common (structural) properties. A node class determines the attributes which all nodes of this class possess, and the relations in which they may participate. In order to factor out common properties, node classes are organized into an inheritance hierarchy. A subclass inherits from its superclasses all attributes defined in the superclasses, and all relations in which nodes of the superclass may participate. Furthermore, it may introduce or redefine evaluation rules for inherited derived attributes (dynamic binding), and it may also define additional attributes.

In the hypertext example, *OBJECT* acts as the root of the application-specific class hierarchy (in fig. 3, node classes and inheritance relations are represented by boxes and solid arrows, respectively). On the next level, we distinguish between *COMPOSITE* and *COMPONENT* nodes which serve as source and sink of *Contains* edges, respectively. For example, the textual declaration of *COMPOSITE* reads as follows:

```
node_class COMPOSITE is_a OBJECT;
    intrinsic Name : string;
end;
```

This means that *COMPOSITE* inherits all properties from *OBJECT* and in addition defines an intrinsic, single-valued *Name* attribute of type *string*. On the third level of the hierarchy, the

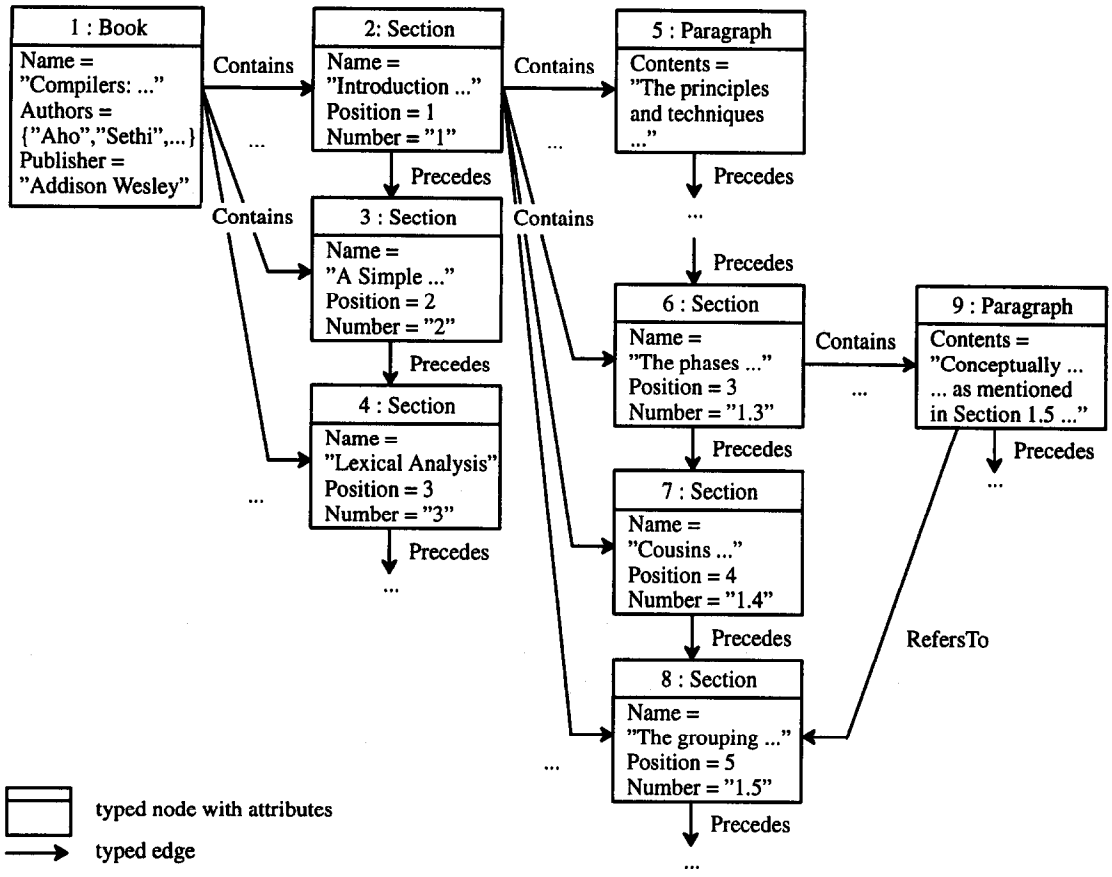


Fig. 2: Graph representation of the structured text shown in fig. 1

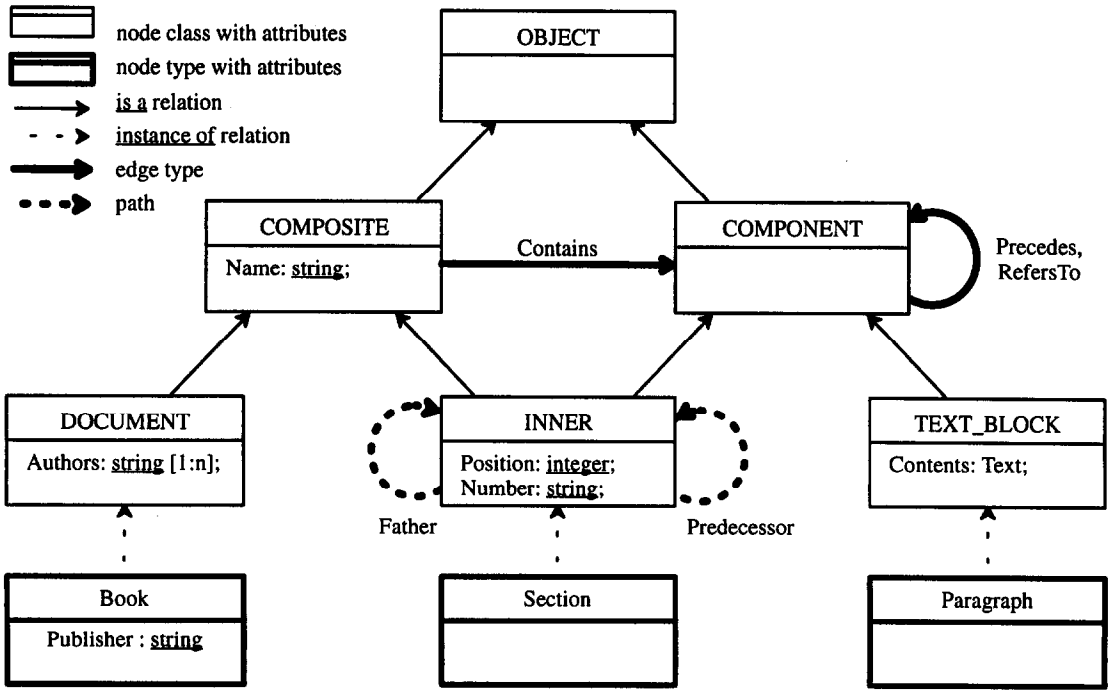


Fig. 3: Graph scheme for the hypertext example

classification is refined such that root, inner, and leaf nodes of the composition tree are distinguished (classes *DOCUMENT*, *INNER*, and *TEXT_BLOCK*, respectively).

A **node type** declaration (bold-faced boxes in fig. 3) syntactically differs from a node class declaration only by its keyword and the replacement of *is_a* with *instance_of*, e.g.

```
node_type Book instance_of DOCUMENT;
    intrinsic Publisher : string;
end;
```

However, it introduces an instance of a class rather than a subclass. *instance_of* and *is_a* relations are related in the following way:

$$(T \ \underline{instance\_of} \ C) \ and \ (C \ \underline{is\_a} \ D) \Longrightarrow (T \ \underline{instance\_of} \ D)$$

This means that a node class $D$ may be identified with the set of all node types which are instances of $D$ or of one of its (transitive) subclasses. Therefore, the instance of hierarchy of the stratified type system may be cut off at the class level (a formal set-valued node type parameter may be replaced with an actual parameter denoting a class, i.e. the set of node types belonging to this class).

**Edge types** represent intrinsic, binary, and directed relations between nodes of certain classes or types. In fig. 3, edge types are represented by solid, labeled arrows. The declaration of an edge type defines source and sink class/type, as well as its cardinality. For example,

```
edge_type Contains: COMPOSITE [1:1] -> COMPONENT [1:n];
```

defines edge type *Contains* which leads from *COMPOSITE* to *COMPONENT* nodes (more precisely: the types of nodes acting as source and sink of *Contains* edges must be instances *of COMPOSITE and COMPONENT*, respectively). Starting from a *COMPOSITE* node, *1* to *n* *COMPONENT* nodes are reached by traversing *Contains* edges. Conversely, each *COMPONENT* node is attached to exactly one *COMPOSITE* node.

**Paths** represent derived relations which are calculated from edges and node properties. Paths are specified by path expressions which are composed of operators for edge traversals, loops, sequences, transitive closures, type/value-based restrictions, etc. In general, paths may be nested, i.e. a path expression may refer to another (or even the same) path expression. Paths provide a convenient and powerful way to define abstract views on graph structures. Applications using paths need not know how the paths are actually calculated.

In our example, two paths are defined which are used in the rules for derived attributes *Position* and *Number* (see below). In fig. 3, paths are represented by dashed arrows in bold face. The path *Father* connects nodes of class *INNER* and is specified as follows:

```
path Father: INNER [1:n] -> INNER [0:1] =
    <-Contains- & instance_of INNER
end;
```

Starting from *an INNER* node, the incoming *Contains* edge is traversed in backward direction. In general, the resulting node only belongs to class *COMPOSITE*. Therefore, a restriction is added which checks whether the node is of class *INNER*. This restriction is needed in the derivation rule for attribute *Number* to ensure that the father node carries a *Number* attribute (see below). *&* is a binary operator on paths which denotes a sequence: Firstly, the *Contains* edge is traversed; subsequently, the node class restriction is applied.

Similarly, the path *Predecessor* is specified:

```
path Predecessor: INNER [0:1] -> INNER [0:1] =
    <-Precedes- & { not instance_of INNER? <-Precedes- }
end;
```

*Predecessor* leads to the first preceding *INNER* node, if any. In general, the graph scheme allows for arbitrary sequences of *INNER* and *TEXT_BLOCK* nodes (e.g. section *1* of the sample book starts with an introductory paragraph followed by subsections, see fig. 1 and 2). Therefore, an iterator (denoted by {...}) is needed to skip all predecessors not belonging to class *INNER*. The iterator is terminated as soon as the condition before the question mark *?* is violated. The result yielded by the iterator is an *INNER* node which therefore carries a *Position* attribute (see below).

The values of **derived attributes** are calculated from the values of other attributes which either belong to the same node or to neighbor nodes. Derived attributes allow for functional specification of attribute values. They relieve applications from the tedious task to keep attribute values consistent after having performed graph modifications. In general, neighbor nodes need not belong to the 1-context; rather, they need only be connected via a path which may be arbitrarily long. In this way, management of derived relations and attributes is fully integrated (note that restrictions in path expressions may access derived attributes).

In our example, attributes *Position* and *Number* of class *INNER* are specified using paths *Father* and *Predecessor*:

```
node_class INNER is_a COMPOSITE, COMPONENT;
    derived Position : integer = [ self.Predecessor.Position + 1 | 1 ];
    derived Number : string =
        [self.Father.Number + "." + string(self.Position) |
        string(self.Position)];
end;
```

In both evaluation rules, conditional expressions are used which are indicated by square brackets *[...]*. A conditional expression consists of a sequence of alternatives which are separated by vertical bars |. When a conditional expression is evaluated, the first alternative is chosen which yields a defined value. In case of the *Position* attribute, the value is either obtained by incrementing the position of its predecessor, or is set to *1* if no predecessor exists. Analogously, the value of the *Number* attribute is calculated by concatenating (operator +) the number of the *Father* node, a dot, and the string representation of its own position, or by merely converting the *Position* attribute if no father exists.

## 3. ARCHITECTURE: THE KERNEL

In the next two sections, we discuss the internal architecture of the GRAS system. In order to provide for re-usability and flexibility, we have designed a **layered software architecture** which is divided into two parts:

- Its lower part (the kernel), which is depicted in fig. 4 and will be explained in this section, consists of layers of data abstraction. Each layer has its own, carefully designed data model and supports the implementation of a wide spectrum of different applications (with different data models) on top of it.
- By way of contrast, all layers of the upper part, which will be explained in the following section, rely on a common, graph-oriented data model. Each layer incrementally adds a set of logically related services to the functionality of the layer beneath it.

When proceeding through the layers of the architecture, we describe their functionality (i.e. the interface provided to upper layers) as well as their realization. We motivate our central design decisions not only with respect to individual layers, but also with respect to their arrangement (i.e. their position within the architecture).

### *3.1. GraphStorage layer*

The *GraphStorage* is the topmost data abstraction layer. It provides a **graph-oriented interface** consisting of GRAS resources (procedures) for creating, deleting, and accessing all kinds

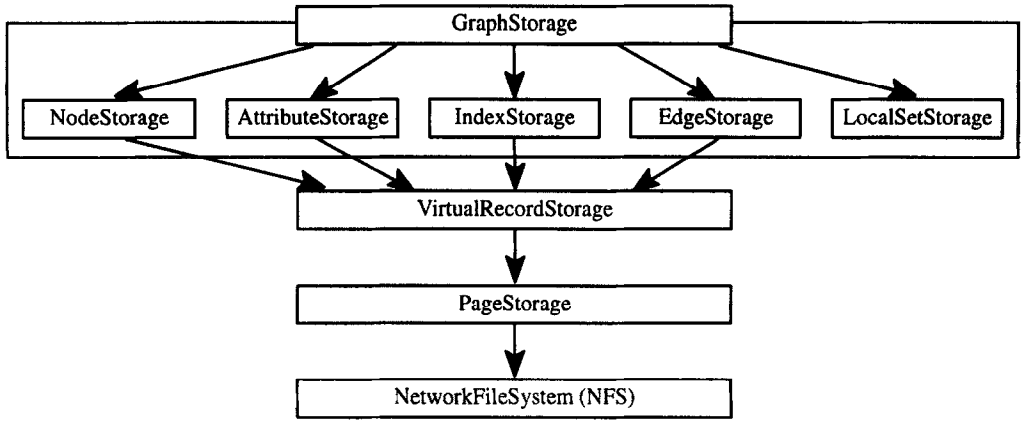


Fig. 4: Lower layers of the GRAS architecture

of graph components. In particular, the interface offers write operations for creating and deleting typed nodes, for creating and deleting typed directed edges, for initializing and modifying arbitrarily long node attributes, and finally for maintaining attribute indexes.

To characterize all read operations, graphs are considered sets of tuples of the following three forms:

- *node := (node id, node type).*
- *edge := (source node id, edge type, sink node id).*
- *attribute := (node id, attribute name, attribute value).*

The GRAS system supports almost all possible **partial match queries** for the relations enumerated above[†],e.g. for determining the in- and out-context of a node, for retrieving all nodes with a certain attribute value, and for returning the names of all attributes which have defined values for a certain node. The results of such partial match queries are either single elements or sets of elements or even binary relations (e.g. the out-context of a node is a binary relation which consists of pairs *(edge type, sink node id)*). Therefore, the GRAS system offers facilities for efficiently handling (ordered) temporary sets and relations.

Internally, each graph is stored in a data structure which is called graph base in the sequel. Each graph base consists of **separate storages** for each kind of data: The *LocalSetStorage*, the only storage which contains volatile data, stores and processes temporary query results (sets and binary relations). The *NodeStorage* has been designed as a repository for nodes themselves and for their most frequently accessed, rather short attributes. Long node attributes (byte sequences of almost arbitrary length) are mapped onto the *AttributeStorage*. The *IndexStorage* maintains maps of attribute values to node identifiers and executes corresponding partial match queries. Finally, the *EdgeStorage* creates/deletes edges and executes partial match queries with respect to these edges. In order to support bidirectional traversal, the *EdgeStorage* stores for each edge both a forward and a backward access path.

Storing different kinds of data within different storages has been one of our most important design decisions. In this way, data necessary for processing different kinds of application requests are stored on different disk pages.[‡] In particular, the separation of large attributes from all other kinds of data considerably speeds up structure-oriented queries. And keeping relations between objects (nodes) separately is an approved technique for accelerating navigational queries (in contrast to "path indexes" in [47] and "access support relations" in [46], GRAS even avoids duplication of intrinsic relations within an additional index structure).

[†]Partial match queries of the forms *(?, edge type, ?)* and *(?, ?, attribute value)* are not supported.

[‡]Clustering within one storage will be discussed below.

On the other hand, operations which affect a node including all its attributes and edges (as creation and deletion) need access to a greater number of pages. This disadvantage seems to be negligible if the database system's cache has a reasonable size. In this case, it makes no difference whether all needed nodes are stored on $n1$ pages, their attributes on separate $n2$ pages, and their edges on separate $n3$ pages or whether nodes are stored together with their attributes and edges on $n1 + n2 + n3$ pages (see also benchmark results in section 5).

### *3.2. VirtualRecordStorage layer*

All permanent storages described in the previous section share the following characteristics:

- They contain persistent data of dynamically varying size which ranges from a few hundred bytes up to some megabytes.
- Each data portion stored in one of these storages has to be identified by a unique identifier. These identifiers are internally used for establishing cross-references between different portions of data. They are also used for representing nodes in application-specific data structures. Unique identifiers must not vary during the whole lifespan of the data portions they belong to.
- Furthermore, some of these storages have to support efficient retrieval of data portions, selected by (only a part of) their identifiers (partial match queries).

Having these common characteristics in mind, we have implemented one parametric *VirtualRecordStorage* with a **record-oriented interface**. This storage realizes all specialized permanent storages mentioned above. The parameters mainly define the structure of records. Any record has an identifier of fixed size and additionally may consist of a data area of fixed size (e.g. for storing node types), a number of data areas of dynamically varying size (e.g. for storing long attributes), and a number of areas for ordered sets of identifiers (e.g. for storing references to other records).

Efficient operation of the GRAS system heavily depends on a careful implementation of the *VirtualRecordStorage*. When designing its storage structure, the following requirements of engineering applications have to be taken into account:

- The storage structure has to provide for efficient access to medium-sized graph bases (with each graph base having the size of a typical engineering document—e.g. a program module—which typically ranges from some kilobytes to a few megabytes).
- The storage structure shall efficiently support editing operations on engineering documents. Therefore, it has to accommodate dynamically growing and shrinking graph bases.
- Finally, clustering of data has to be supported in order to provide for efficient access to logically related data.

In order to meet these requirements, we have designed an indexing scheme based on a combination of tries and static hashing. Unique identifiers are mapped onto physical addresses in the following way: A special variant of binary index trees, so-called "tries" (cf. e.g. [25]), is used to select a record's page, and static hashing determines an appropriate position on a selected page. The algorithms which generate identifiers for new records (and thus determine the page positions of new records) are sensitive to requests of the application layer. The application may specify neighborhoods for records which are often accessed together (e.g. nodes connected by certain edges).[†] This information is used for clustering logically related records onto adjacent storage locations. Since graph bases may change rapidly and access patterns may also undergo changes due to different kinds of applications, the **clustering algorithm** operates heuristically and incrementally and causes only a negligible overhead. It may be supplemented with sophisticated and time-consuming batch-oriented clustering algorithms (see e.g. [6, 33, 77]) on top of the GRAS system. A more detailed discussion of the *VirtualRecordStorage* is beyond the scope of this paper; the interested reader is referred to [48].

[†]The application specifies neighborhoods for node records only. The distribution of edges and node attributes over pages is the same as for the nodes they belong to.

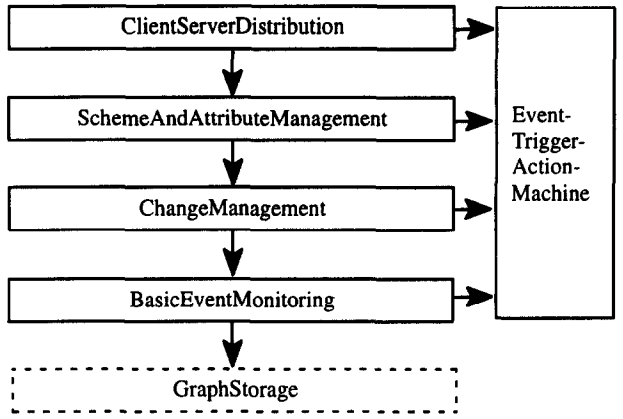


Fig. 5: Upper layers of the GRAS architecture

### *3.3. PageStorage and NetworkFileSystem layers*

The *PageStorage* provides a **page-oriented interface**; each page is a sequence of bytes which has a fixed length. Graphs are not required to fit into main memory; therefore, the page storage maintains a page cache in main memory and controls transfer of pages between disk and main memory. Paging is driven by an LRU strategy which takes additional factors such as frequency of access and priority into account (log pages are assigned the highest priority, followed by index pages and data pages). The size of the page cache may be fixed as needed. If the underlying operating system provides a sufficiently large virtual memory, the size may be chosen so that all graph operations are eventually performed in virtual memory. In this way, paging is delegated to the operating system.

Finally, the *NetworkFileSystem* layer provides the interface to the underlying operating system's **distributed file management**. It maps page sequences onto files. Since its interface is independent of a particular operating system, it is an easy task to port the GRAS system to another operating system. Only a few pages of source code implementing the file system layer have to be adapted or written anew.

## 4. ARCHITECTURE: ENHANCING LAYERS

All layers on top of the GRAS system's kernel *GraphStorage* (see fig. 5) rely on a common, graph-oriented data model. Each layer incrementally adds a set of **logically related services** to the functionality of the layer beneath it. In terms of object-oriented programming, the architecture of the upper part consists of an inheritance hierarchy where each layer is identified with a graph class. In contrast to this, layers of the lower part rely on different data models and are related by 'import' rather than 'is_a' relations.

The layers of the upper part introduce event handling (*BasicEventMonitoring*), transactions, undo/redo, and deltas (*ChangeManagement*), graph schemes and management of derived attributes and relations (*SchemeAndAttributeManagement*), and concurrency control and distribution (*ClientServerDistribution*). Orthogonal to these layers, an *EventTriggerActionMachine* provides common functions for event handling which are used in all layers. Note that event handling cannot be introduced once and for all in *BasicEventMonitoring*; rather, it has to be extended as new functionality is added in higher layers.

Arrangement of upper layers will be motivated at the end of each subsection, respectively. Since at first glance the extensions provided by these layers seem to be independent of each other, it is attempting to follow a 'toolkit' approach: Firstly, for each extension a corresponding subclass of GraphStorage is provided; secondly, all desired combinations are realized by multiple inheritance. However, a severe drawback of this approach consists in the fact that one extension cannot be

implemented with the help of another (e.g. management of derived attributes cannot exploit event handling). Furthermore, combination of different extensions is not for free and may involve a great deal of re-implementation.

### *4.1. BasicEventMonitoring layer*

Event/Trigger mechanisms are a fundamental concept of most so-called **active database systems** [8, 18, 24]. They are also used in modern applications for separating the user interface component from the functional part of the application. Event/Trigger mechanisms may be used for incremental supervision of certain consistency constraints, incremental computation of derived data like derived attribute values or relations, and a-posteriori integration of different applications accessing the same data structure.

In GRAS, **events** refer to database transitions, i.e. atomic changes to a graph database. Roughly, these database transitions correspond to the write operations introduced in section 3.1. In some cases, however, a write operation is mapped onto multiple database transitions (e.g. deletion of a node includes deletion of all adjacent edges). Since an application is interested only in a specific subset of all evoked events, it may define event patterns. Furthermore, **triggers** are used to perform bindings between event patterns and corresponding **action routines**. In general, m:n bindings may be established. Triggers may be activated and deactivated at any time while a graph is open. When several triggers fire for a database transition, the execution order can be determined using trigger priorities. When an event matches an event pattern of a trigger, the action routine is called with actual parameters which depend on the type of event pattern (e.g. the node to be deleted, the source node of a new edge, etc. ).

GRAS supports various **event types** which reflect both the data model and the set of change and administration operations. This includes e.g. event types for

| | |
|---|---|
| nodes: | *(node type, node id, {Create, Delete})* |
| edges: | *(src node type, src node id, edge type, sink node type, sink node id, {Create, Delete})* |
| attributes: | *(node type,node id, attribute name, {Init, Read, Write})* |

Beside these specific components, events also have a set of common components which describe further properties. This includes flags which indicate whether the system is just before or after the corresponding state change (pre/post events), whether the event was produced as a side effect of a state change or by direct user invocation (natural/synthetic events), and whether the corresponding state change was produced by a normal application operation or came from the replay of undo/redo logs (primary/replay events, see next subsection).

For each event type there exists a corresponding **event pattern type** which allows to replace arbitrary event components with the special wildcard value *Any*. For example, the event pattern *($NT_1$, Any, Create)* matches all events resulting from creation of a node of type $NT_1$, whereas *(Any, Any, Any, Any, 100, Delete)* matches all events occurring from the deletion of an incoming edge of node *100*.

Event management is realized using a generic **event-trigger-action machine** (programmed in C++) (see e.g. [77] for a general discussion about such machines), which is instantiated by GRAS-specific event patterns and corresponding compare operations. Relevant state changes in GRAS are transformed into events and sent (via procedure calls) to this instantiated machine, whereas the definition of event patterns, triggers and actions are performed by the application. The machine also allows for the definition of persistent event patterns, triggers, and actions, which are automatically activated when accessing the graph. In this way, dynamic constraints may be enforced in addition to static constraints of the graph scheme.

As event management is based on the concept of monitoring state changes to the database kernel, the layer evoking events due to database transitions has to be as close as possible to the kernel. Otherwise, database operations performed by layers between the kernel and the event management layer would be undetectable. For other event classes concerning e.g. change management or concurrency control which do not necessarily result in database transitions, corresponding events

must be evoked directly from within these layers. This results in an architecture with a basic event management layer placed above the kernel, and an event-trigger-action machine standing vertically aside all enhancing layers.

### *4.2. ChangeManagement layer*

The *ChangeManagement* [79] layer is placed above *BasicEventMonitoring* and adds to this layer all functions for logging, storing, and executing sequences of change operations. Change management provides for user recovery (undo/redo of user commands), system recovery (recovery from system failures), and deltas (efficient storage of versions). These tasks are handled by GRAS in an integrated way by maintaining logs of change operations.

Following [5], we define user recovery as recovery actions which are controlled by the user rather than by the system which he uses. For example, the user may undo a command which was activated inadvertently or switch back and forth between breakpoints when debugging a program. GRAS supports implementation of user recovery in the following way: While a graph is open, an application may define checkpoints. Typically, this is done when the execution of a user command terminates. By means of **undo** and **redo** operations, the application may switch back and forth between arbitrary checkpoints (within the limits of the current session initiated by *OpenGraph*). Checkpoints are ordered sequentially. Undo and redo are constrained such that they always yield a semantically consistent state; this could not be guaranteed if e.g. selective undo were supported (undo command $i$, but not commands $i+1, \ldots, n$).

Checkpoints are also used for system recovery: On system failure, GRAS tries to restore the most recent checkpoint. Furthermore, system recovery is supported by **nested transactions** which are useful for implementing user commands in a layered architecture: Each application layer defines a corresponding level of consistency and uses transactions to guarantee atomicity of the operations which it provides. In particular, each layer is able to abort operation sequences of the next lower level which lead to an inconsistent state from its point of view. Note that checkpoints are treated as boundaries of top-level transactions.

In addition to supporting undo/redo "in the small", a software engineering environment has to provide for undo/redo "in the large". This is the task of version control [76]. While GRAS does not incorporate a specific model of version control (which is nearly always subject to debate), it does provide flexible mechanisms for efficient storage of versions by **graph deltas** [79]. Here, the term "delta" denotes a sequence of operations that, being applied to a version $v1$, yields another version $v2$. Instead of storing all versions of one document completely, it suffices to store a few of them completely and reconstruct the others by means of deltas.

GRAS provides for flexible delta control: Deltas save storage space at the cost of access time. Before a version may be operated on, it may have to be reconstructed. Thus, (optional) use of deltas is controlled by the application in order to achieve an appropriate balance of storage and runtime efficiency. Thereby, the application may choose between **forward** and **backward deltas**. This is illustrated in fig. 6 which shows a version tree (part a) and different examples for the storage of its four graph versions (parts b-d):

- Only forward deltas are used (part b).
- The most recent version on the main trunk is stored completely; all other versions are reconstructed by means of backward and forward deltas (part c). This solution was pioneered by RCS [76].
- Only backward deltas are used (part d). In this case, there may be multiple ways to reconstruct a version. Then, GRAS selects the most efficient one automatically.

Virtually all functions of change management are implemented in a uniform way by **logs** of change operations on graphs. Two kinds of logs are needed: The forward log is used to implement

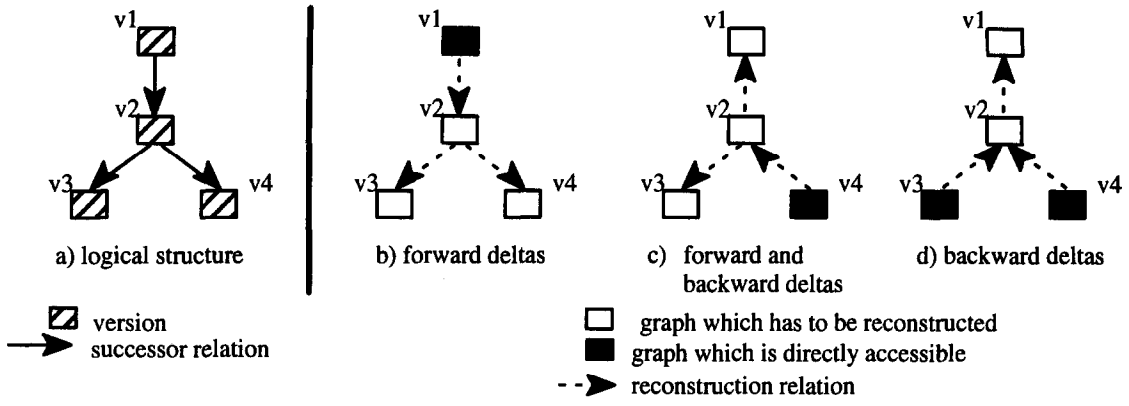


Fig. 6: Flexibility of delta control

redo, forward deltas, and recovery from system crashes,† while the backward log is used analogously for undo, backward deltas, and transaction abort. Thus, logs are reused in an elegant manner for multiple purposes. Particularly, logging of change operations yields a delta on the side so that its costly a-posteriori reconstruction may be avoided. However, direct use of logs may be inefficient because the effect of one operation may be overridden by subsequent operations. Therefore, logs are compressed a-posteriori by removing redundant operations.

To conclude this subsection, let us comment on the position of the *ChangeManagement* layer within the GRAS architecture: In order to keep deltas short, *ChangeManagement* has been placed in the upper part of the GRAS architecture which relies on a common graph-oriented data model. Logging of graph-oriented operations is much more space-efficient than logging on any layer below the *GraphStorage*.

Originally, *ChangeManagement* was placed below *BasicEventMonitoring* for the following reason: If events are raised during execution of the undo log, it cannot be guaranteed in general that the corresponding action routines do not modify the graph. Such modifications would lead to a state which in general is different from the state to be reconstructed. However, this approach has the following disadvantage: applications which do not support undo/redo themselves are not able to react on changes caused by undo/redo.

Therefore, *BasicEventMonitoring* and *ChangeManagement* have been switched recently. In order to cope with undo/redo, event handling is extended in the following way:

- Undo/redo events are introduced (in the *ChangeManagement* layer). Such events are used by applications featuring an undo/redo facility.
- Each event handler has to declare whether its action routine is to be invoked during undo or redo. If activation is required, the action routine must not modify the host graph (rather, it is used for updating dependent information outside the host graph).

### *4.3. Scheme- and AttributeManagement layer*

Presenting the GRAS system's basic graph operations, we have neglected one important requirement. The system has to preserve a graph's consistency with respect to its graph scheme by rejecting forbidden graph modifications and by recomputing derived attribute values. Concerning the hypertext example of section 2, all operations creating e.g. two emanating *Precedes* edges at one *COMPONENT* node must be rejected, and insertions or deletions of *INNER* nodes eventually trigger the reevaluation of dependent *Position* and *Number* attributes.

†For system recovery, forward logs are supplemented by a shadow page mechanism in the *PageStorage* layer which prevents any corruptions of an open graph's initial state. After "hard" system crashes, the most recent consistent graph state is reconstructed by applying the forward log to the still existing initial graph state until the last recorded checkpoint.

In order to be able to fulfill these tasks, information about a graph's class hierarchy, its attribute dependencies and evaluation functions, etc. must be available at runtime. Therefore, GRAS supports the construction of internal graph schemes which provide all these informations in an efficiently accessible format. Such a graph scheme may be extended arbitrarily during its whole lifetime so that the upward-compatibility of graph-based tools and their data structures is guaranteed.

Discussing the realization of this part of the GRAS architecture, we focus on its most important task: the incremental evaluation of derived attributes. For this purpose, we have implemented a variant of the well-known **two-phase, lazy attribute evaluation algorithm** [41] which has already been used successfully within another graph-oriented database management system (Cactis [42, 43]). This demand-driven algorithm uses a potentially cyclic static attribute dependency graph containing all information about possible attribute dependencies and evaluation rules. It works as follows:

**Phase 1:** The assignment of a new value to an intrinsic attribute or the insertion/deletion of certain edges triggers the **invalidation** of all potentially affected derived attributes (propagation stops at already invalid attributes).

**Phase 2:** The reevaluation of an invalid attribute will be delayed until the first attempt to read its value. During its then necessary **reevaluation**, read accesses to other attributes' values may raise evaluation processes for these attributes, too (a bookkeeping mechanism guarantees the abortion of the attribute evaluation process in the presence of forbidden cyclic attribute dependencies).

This rather primitive two-phase algorithm is at least equivalent (with respect to the number of attribute reevaluations after graph updates) to all other incremental attribute evaluation algorithms if almost all graph (attribute) changes result in changes of all potentially affected attributes, or a graph contains many rarely accessed attributes with often changing values.

Note that both conditions are fulfilled in the case of our hypertext example (and in many other cases, too). Each insertion or deletion of a section changes *Number* and *Position* attributes of all following sections and their subsections, but only *Number* and *Position* attributes of currently displayed sections (and their predecessors) must be up-to-date. For a more detailed discussion of the advantages and disadvantages of different graph-based incremental attribute evaluation algorithms, the reader is referred to [2, 71].

During the adaptation of the algorithm to the special needs of GRAS we encountered one major problem which has not yet been addressed by the graph attribute evaluation algorithms of [2, 42, 43]. The language PROGRES allows for the definition of n-context attribute dependencies: An attribute *A1* of a node *N1* may depend on an attribute *A2* of a node *N2*, where *N1* and *N2* are connected via a path of arbitrary length $n$ ($n$ edges have to be traversed to reach *N2* from *N1*). Therefore, graph modifications at remote locations may influence an attribute's value. The hypertext specification of section 2 contains an example for the usefulness of n-context attribute dependencies: the *Position* of a *Section* will be computed by incrementing the *Position* of its preceding *Section* (or by assigning the value *1* in the case of the first *Section*). In order to find this preceding *Section*, we have to follow a potentially unrestricted number of *Precedes* edges from sinks to sources, skipping all *COMPONENTS* without a *Position* attribute (i.e. *Paragraphs* and *Subparagraphs*).

In order to be able to handle these remote attribute dependencies, we have introduced the concept of **virtual attributes** (cf. [71]) in GRAS. These attributes are invisible to application programs, and they do not even possess values on their own (with exception of an invalid flag). Their only purpose is to reduce the difficult problem of propagating invalid flags in the case of n-context dependencies to the already solved 1-context problem. In the case of our *Documentation* example we have to introduce one virtual attribute *VAtt* which propagates changes of *Position* attributes across an arbitrary number of *COMPONENTS* without *Position* attributes. Consider e.g. the deletion of the first *Section* in fig. 7. This operation initiates the following propagation process:

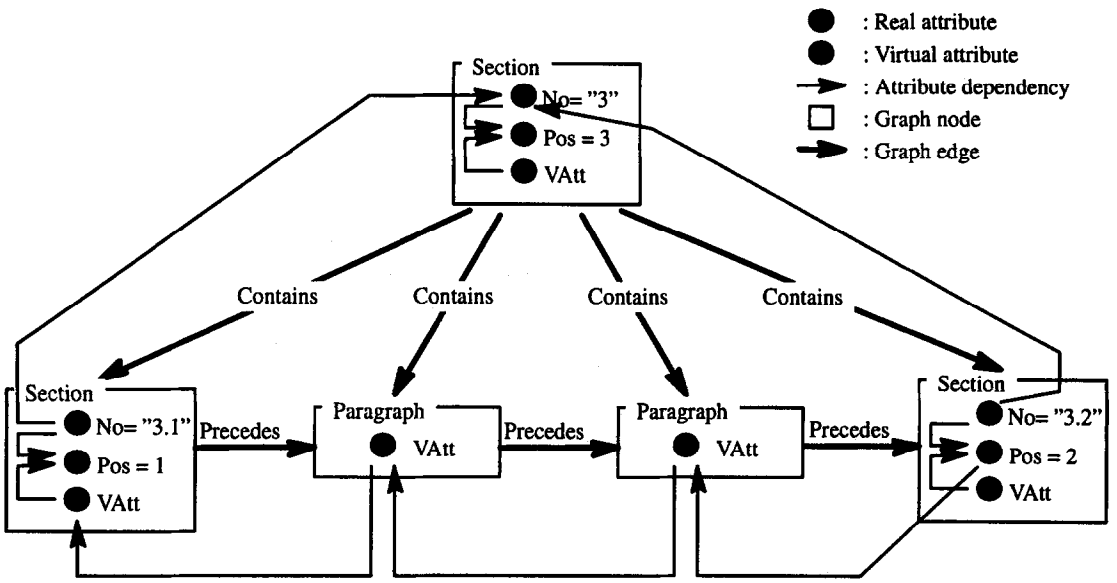


Fig. 7: Cut-out of a documentation's actual attribute dependency graph

1. the *VAtt* attribute of the first *Paragraph* is invalidated because its incoming *Precedes* edge is deleted,
2. this invalidation propagates to the *VAtt* attribute of the next *Paragraph*,
3. the *Position* attribute of the next *Section* is invalidated in response to the invalidation of the *VAtt* attribute of the previous *COMPONENT*,
4. and *Number* and *VAtt* attributes at the same node are invalidated according to their dependency on the *Position* attribute.

All these attributes remain invalid as long as their values are not required by the application, i.e. they will be recomputed on demand (and propagation processes of subsequent insertions or deletions immediately stops at already invalid attributes). Reading e.g. the *Number* attribute of a *Section* has the side effect of recomputing *Number* and *Position* attributes of all preceding *Sections* and of "validating" the *Vatt* attribute of all preceding *Paragraphs*. Similarly, accessing the index of a derived attribute $A$ (not present in our example) triggers the reevaluation of all invalid $A$ attributes of a graph. Otherwise, the GRAS system would not be able to answer queries like "find all nodes with a derived attribute $A$ which has the value $V$".

Last but not least it was a straightforward idea to use the same concepts and techniques for the incremental computation of derived relations, which are (recursively) defined by means of complex path expressions. By viewing them as a special form of derived node reference attributes, our attribute evaluation algorithm is able to materialize their values on demand.

In this way, GRAS is able to efficiently maintain derived information about graphs in a similar way as e.g. in the object-oriented database system GOM [33, 46]. But note that the actual attribute (and relation) dependencies of our sample graph in fig. 7 do not really exist in the form of additional relations, as in the system GOM, but will be deduced during the propagation process by means of a **static attribute dependency graph** (for further details see [48]).

We conclude the discussion of the *SchemeAndAttributeManagement* layer with a final remark about its location in the GRAS architecture on top of the *ChangeManagement* layer. This position has the advantage that undo/redo handles derived and intrinsic data in a uniform way, i.e. all computed derived data is still available after undo/redo operations. By exchanging the positions of both layers, we could follow the approach of Cactis [41, 42]. Cactis does not log values of derived data, but invalidates them during undo/redo and recomputes them on demand. While the position

of the *SchemeAndAttributeManagement* layer on top of *ChangeManagement* is discussable, there are no doubts about its cooperation with GRAS' *EventTriggerActionMachine*. New resources for manipulating graph schemes and derived data result in new types of events and event patterns that must be monitored and handled within this machine.

### *4.4. ConcurrencyControl and Distribution layer*

The top-most architectural layer of GRAS handles concurrency control and distribution. As described in section 3, GRAS has a two-layered object model: On the fine-grained level, the application operates on nodes and edges. On the coarse-grained level, graphs as structured collections of nodes are manipulated. Both concurrency control and distribution refer to the coarse-grained level, i.e. graphs rather than nodes are locked and distributed. Note that this coarse-grained approach fits well with our specification paradigm outlined in section 3: operations such as graph rewrite rules are attached to graphs rather than single nodes and edges. Locking or distributing single graph elements would be fairly complex and would incur a high overhead, which in particular is not justified when taking typical access patterns in software engineering environments into account (see section 5).

Distribution is realized by a variant of the client-server approach used by most distributed database systems [20, 22]. However, instead of using one centralized database server, GRAS uses a **pool of graph servers**. Each of these graph servers controls and manipulates one or more graphs. Whenever an application requests access to a graph, it is connected to a graph server which performs all of its requests for this graph. This graph server is selected according to following rules:

1. If the graph is already open due to a request from another application, the corresponding graph server is used.

2. If a graph server exists which is willing to accept another graph, this server is selected. Acceptance is calculated within the servers based on current host load and number of served graphs.

3. Else a new graph server is started. The location of this server is determined by the current load of a set of trusted hosts. It is even possible to start more than one server on a host.

A **special control** server—not depicted in fig. 8—keeps track of the mapping of open graphs to graph servers. This control server also serializes open and close requests for graphs to avoid race conflicts and interconnection between different graph servers. Fig. 8 shows a typical scenario with three client and two graph server processes (without the control server). Notice that both $Client_2$ and $Client_3$ access the same graph $G_3$, and that $Client_2$ communicates with more than one server.

The interoperability of clients and servers is realized by inserting a communication layer consisting of two parts: the client side communication interface (CSCI) and the server side communication interface (SSCI). The procedural database interface as described in the previous sections is offered to clients through the CSCI, which uses a **remote procedure call** library to forward calls down to the SSCI of the appropriate server. These interfaces are also responsible for concurrency control and error handling. Furthermore, they handle communication failures and client or server shutdown using follow-up-RPCs and heartbeat protocols.

To organize concurrent accesses of applications to the same graph, GRAS supports an **access group model**: Each application accessing a graph belongs to a group which has an own copy of the accessed graph. Groups are temporary collections of applications which access the same graph. Groups are created explicitly by one application (which becomes the initial member of this group), potentially joined by other applications (which want to share their view of the graph with other members of that group), and terminate implicitly when the last group member leaves it (by closing the graph).[†]

[†]This is analogous to the UNIX file access model where a file is physically closed when the last accessor (using the same handle) closes the graph.

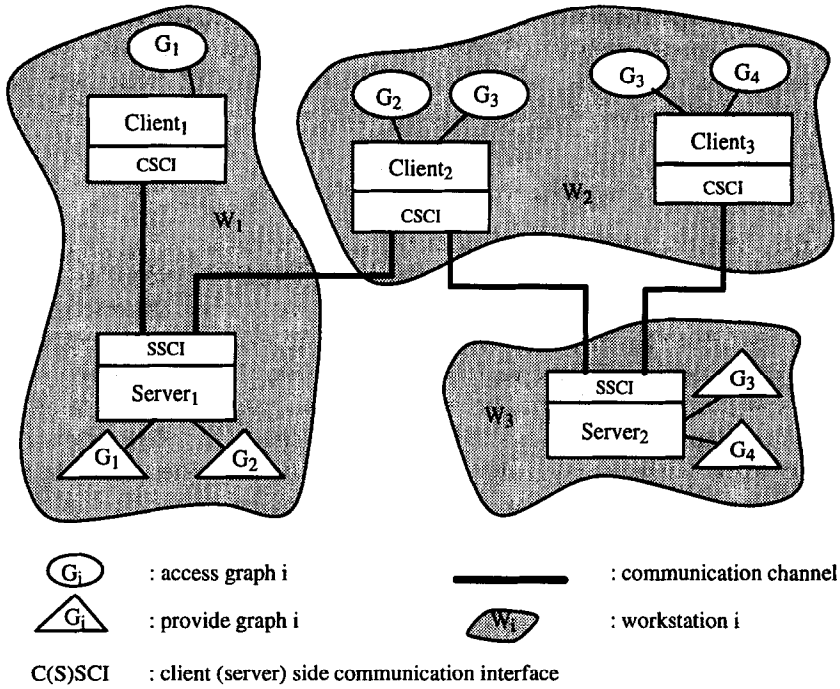


Fig. 8: A typical distributed GRAS scenario

Concurrency of members within a group is controlled by the interaction granularity attribute determining exclusive access units within the time space. Access conflicts due to concurrency conflicts are solved by either delaying or aborting the execution of the conflicting operation (the actual reaction is eligible at operation invocation). GRAS supports three different levels of **interaction granularity**:

- **Operational access** provides for a graph lock during the execution of one basic GRAS operation. This is the lowest level of locking over time and, therefore, yields the highest possible rate of concurrency. Allowing even concurrent execution of these operations is possible (although not yet implemented), but should be transparent for the application.

- The next level, **transactional access**, allows application access only within mutually exclusive top-level transactions which perform consistency-preserving transformations. As the whole graph is locked for the time span of the top-level transaction, lock conflicts cannot occur during its execution. This allows the application to perform operations without interfering with other applications. This decoupling of applications can be seen as a great advantage, as they can operate on shared graphs without knowing or even noticing (besides delay of transaction processing) the presence of other applications. These top-level transactions are also the units of undo and redo (cf. section 4.2), because the SSCI automatically checkpoints the graph after each successful top-level transaction.

- Compare this with the **sessional access** mode which reserves the graph for one group member. This mode virtually provides for a single-user database system. Accessing the graph is allowed for exactly one member of the group, and all other members are delayed (or rejected) upon the OpenGraph operation. When the active member leaves the group (by closing the graph), the next pending member of this group becomes the active one.

As groups operate on own copies of the graph, interaction between different groups only takes place when groups terminate. To avoid update conflicts between access groups of the same graph, at most one group is allowed to replace the original version of the graph by its copy on termination (i.e.

to make its modifications permanent). The creation of such a **persistent-modification group** is denied when there already exists a group with this attribute. As non-persistent-modification groups have no permanent effect on the graph, more than one of these groups is allowed to exist at the same time. Applications join groups by specifying their names (supplied at group creation time).

To summarize, *ClientServerDistribution* allows different client processes to concurrently access the same graphs by sending all requests to responsible server processes via rpc-calls. Furthermore, clients may also possess a number of private graph copies (via check-out/check-in) for which they have exclusive access rights and which they are allowed to modify locally without any needs for interprocess communication. The integration of different client processes is supported by event handling facilities. Therefore, the *ClientServerDistribution* layer needs the services of the *EventTriggerActionMachine*, and it distributes events raised by one client to all other clients that are interested in. Finally, *ClientServerDistribution* is realized on top of *SchemeAndAttributeManagement* in order to maximize the complexity of services provided at the GRAS system's interface. This minimizes the number of necessary rpc-calls between client and server processes that are necessary to perform a given task.

## 5. EXPERIENCES AND APPLICATIONS

After having presented the data model and the architecture of GRAS, the current section is devoted to experiences and applications. Firstly, we describe the results of a benchmark which we have carried out to evaluate our design decisions. Subsequently, we discuss how we have used GRAS as a database system for software engineering applications. Finally, we present the PROGRES database development environment which has been implemented on top of GRAS and considerably exceeds the functionality offered by GRAS itself.

### *5.1. Benchmark*

In order to evaluate the performance of GRAS, we have carried out the **hypermodel benchmark**. Both the benchmark and our results will be presented concisely below; for a more comprehensive description, the reader is referred to [4] and [48], respectively. We have selected the hypermodel benchmark because it comprises a large set of operations on a rather complex database. Therefore, it is well-suited to measure the performance of a software engineering database system. In contrast, many benchmarks which have been designed for relational database systems [36] do not meet this requirement. Furthermore, even the OO1 benchmark for object-oriented database systems [13] uses a database which contains only one type of relationship (connecting randomly selected objects). Therefore, it is not appropriate to test the effects of different clustering algorithms with respect to different types of queries.

The data model underlying the hypermodel benchmark is an abstract variant of the documentation example of section 2 (cf. fig. 2). A hypermodel database's dominant structure is a totally balanced tree with fan-out 5. Every node within this tree possesses a couple of integer attributes. Inner nodes are sources of an ordered one-to-many relationship (1n-rel.) connecting any node of level n with its five children nodes at level n+1. Additional attributed many-to-one relationships (m1a-rel.) connect each node of the database with another randomly chosen node of the database.[†] Leaf nodes either carry text attributes (between 10 and 100 words) or bitmaps (between 100x100 and 400x400 pixels). The proportion of text to bitmap nodes is 125 to 1.

The benchmark itself comprises (among others) operations for creating the initial test database with clustering along the one-to-many-relationships and closure operations which follow a specified type of relationships from 50 randomly chosen nodes on level 3 (up to a depth of 25 steps in the case of the potentially cyclic m1a-rel.). In order to be able to measure the effects of caching and clustering, closure operations are performed twice. The first run (cold) starts with empty operating

[†]The hypermodel benchmark introduces even a third kind of mn-relationships which have characteristics somewhere in between 1n-rel. and m1a-rel. For further details about these relationships see [4].

system's file buffers and database caches and the second run (warm) with file buffers and caches whose state is determined by the cold run.

After the brief description of the hypermodel benchmark, we will now present and analyze the most important performance results of its implementation on top of GRAS on the following hardware platform: a Sun 4/390 in multi-user mode (but without any other users logged in) with 32 MB main memory and a 1000 MB CDC IPI 9720 disk under Sun OS 4.1.1. Since the benchmark does not perform concurrent accesses on one graph, the client/server facilities of GRAS were not used at all. On the other hand, the *ChangeManagement* layer degrades performance although its functionality is not exploited by the benchmark.

Fig. 9 presents the disk storage consumption and performance results for three databases of different sizes with each database being five times greater than the next smaller one. All reported times are given in milliseconds and have been divided by the overall number of affected nodes or binary relationships (e.g. the elapsed time for creating 3906 inner nodes of the level 0-6 database is 3906*7 milliseconds). Furthermore, the GRAS system's cache size has been set to the required upper limit, i.e. to 4 MB. Note that the disk storage consumption of each database is directly proportional to its net size. Furthermore:

- Times reported for creating inner nodes (*C.INNER*) are identical for all databases. This is due to the fact that the benchmark starts with the creation of all inner nodes, and that all inner nodes fit into a 4 MB cache.

- Times reported for creating leaf nodes (*C.LEAF*) are always greater than those for inner nodes. This has the following reasons: Leaf nodes possess additional long attributes whose creation is rather time- and space-consuming. Furthermore, each leaf node will be put onto the same page as its already existing inner parent node. On the other hand, long attributes are stored on separate pages. Alternating accesses to *NodeStorage* and *AttributeStorage* pages results in a large number of page faults.

- Creation times for 1n-relationships (*C.1n-rel.*) are very low and remain nearly constant for all databases. This is a result of the design decision to store (long) attributes and relationships (edges) on separate pages. Thus, only a small number of *EdgeStorage* pages is affected. Furthermore, additional node existence checks (to guarantee referential integrity) require only read accesses to a limited number of *NodeStorage* pages. The order of these accesses is even compatible with the chosen clustering strategy.

- Creation times for m1a-relationships (*C.m1a-rel.*) are always greater than those for 1n-relationships, and they are noticeably influenced by a database's size. This is due to the fact that each m1a-relationship is represented by one link node and two additional edges (instead of one edge only). Therefore, this operation demands additional write accesses to old *NodeStorage* and new *AttributeStorage* pages in a completely random fashion. This leads to a significantly greater number of (dirty) page transfers between cache and disk.

Similar explanations hold for the cold/warm closure traversal results:

- Cold traversals (*T.1n*-c* and *T.m1a*-c*, respectively) along 1n-rel. with read accesses to integer node attributes are more expensive for small databases than traversals along m1a-rel. with read accesses to their attributes (probably because the greater number of node attributes is scattered over a greater number of pages than the smaller number of link attributes). This initial effect is soon compensated by the fact that traversals along 1n-rel. are compatible with our clustering strategy. Therefore, the probability for crossing page boundaries is almost independent of the database's size. But traversals along m1a-rel. lead from randomly selected sources to randomly selected sinks and touch a considerably greater number of pages (in this case the probability for crossing page boundaries is proportional to the overall database size).

- The explanations of all warm closure traversal operations (*T.1n*-w* and *T.m1a*-w*, respectively) are trivial: almost all accessed data is already present in the GRAS system's cache.

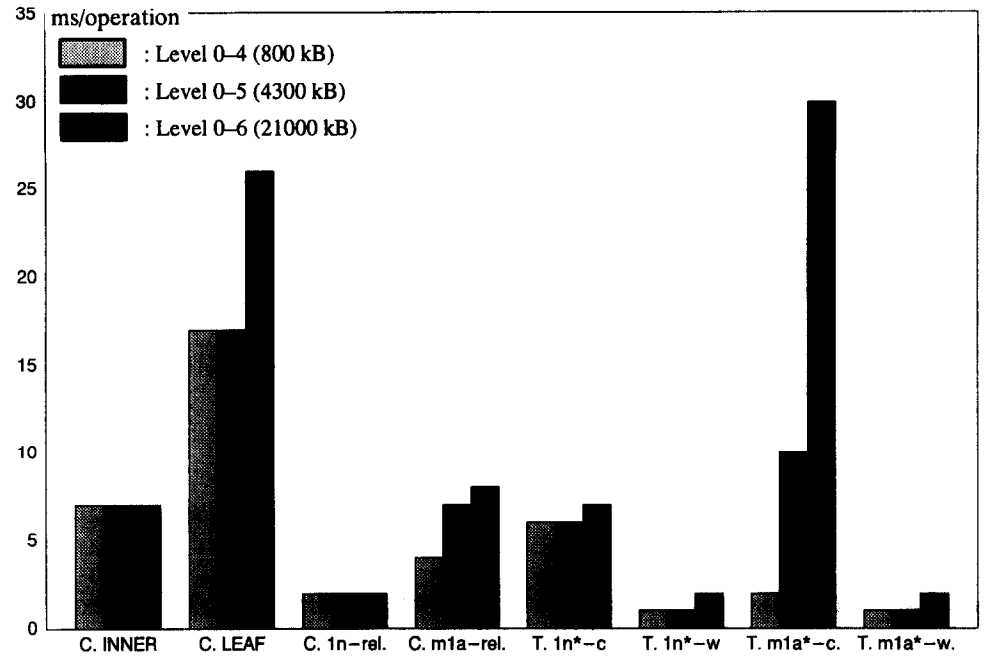


Fig. 9: Benchmark results

Note that in the case of the level 0-6 database with 21 MB disk space requirements only the separation of different kinds of data ensures that almost all accessed data fits into a 4 MB cache (both operations touch about 40% of the nodes of the database).

The presented benchmark results for growing databases and constant cache size are mainly determined by the following design decisions:

- The order in which nodes and 1n-relationships are created is compatible with the system's clustering strategy along 1n-relationships.
- Especially 1n-relationships are concentrated on a small number of pages; this is caused by the system's strategy to store different types of information on different pages.
- The GRAS paging system, which is responsible for the management of all available cache space, uses a priority-based LRU-strategy guaranteeing fast access to all currently used log pages, index pages, and to all frequently and/or recently used data pages.

### *5.2. Software engineering applications*

A first prototype of the GRAS system—described in [10]—was already realized in 1985. Since this time gradually improving versions of GRAS have been used at different sites within the software engineering projects IPSEN [61], Rigi [59], CADDY [26], MERLIN [64], and MELMAC [21]. The following discussion mainly refers to the IPSEN project, but applies more or less also to all other projects listed above.

In order to store all data produced during evolution of a software system, a **project database** is required which is shared among all members of a development team. Such a database is a collection of related **documents** (requirements specifications, software architectures, modules, test plans, etc.). Each document has a fine-grained internal structure which has to be consistent with a corresponding syntax description (database scheme). In our opinion, these documents and not their internal components (increments) are 'natural' objects for distribution, concurrency control, version management, etc.

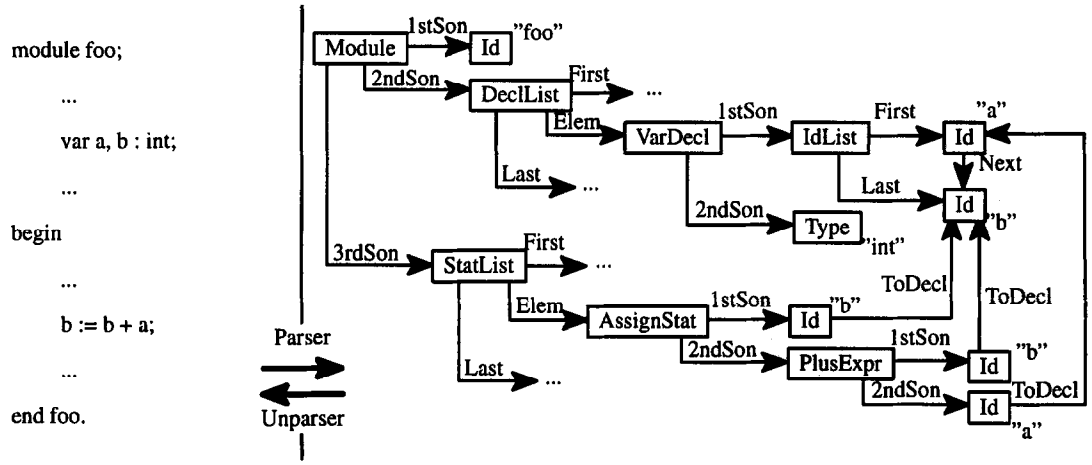


Fig. 10: External and internal representation of a software document

From a user's point of view, a document is a hierarchically structured piece of text (potentially including diagrams etc.), and all document modifications result in modifications of this **external representation**. But from the developer's point of view, a document is a complex graph structure. From his perspective, tools primarily manipulate the underlying **internal representation** which is hidden from the user. Fig. 10 gives an example of an external and an internal representation, respectively. Internally, a document is represented by an **abstract syntax graph** [27], i.e. an abstract syntax tree augmented by context-sensitive edges (e.g. for connecting applied occurrences of identifiers to their declarations). The internal representation is modified by a syntax-directed editor; on the other hand, a text-oriented editor operates on the external representation. Incremental parsers and unparsers keep both representations consistent with each other.

The GRAS data model of directed, attributed graphs allows us to store document representations in a natural, straightforward way, and its graph schemes are the appropriate means for modeling and guaranteeing correctness of documents with respect to their syntax. Context-sensitive relations—which are derived from the context-free structure—may be represented as paths so that GRAS is responsible for keeping them up-to-date. If this solution is considered too inefficient, they may alternatively be represented as edges which are maintained by the application (as shown in fig. 10). Finally, the GRAS system's services may be used to recover from system failures, to abort consistency-violating modifications, and to undo/redo arbitrarily long sequences of edit commands.

Up to now, we have mainly focused on single documents and their representations. But a project database does not merely consist of a set of isolated documents. Rather, documents are connected by manifold relations, e.g. life-cycle dependencies between requirements specifications, software architectures, and module implementations. These relations have to be taken into account when the project database is managed by a software engineering database system.

Fig. 11 illustrates how a software engineering database is represented by a set of interrelated graphs each of which corresponds to a certain document. Documents are arranged in a hierarchy whose leaves are called **simple documents** in the sequel. For example, an architecture graph represents the architecture of a software system, i.e. the set of modules and their export/import relations. For each module whose interface is specified in the architecture, the corresponding implementation is stored in a module graph (an example of which was given in fig. 10).

In order to describe configurations of (versions of) interdependent documents, **configuration documents** are needed which are based on some kind of formal language and are manipulated by structure-oriented tools. Each configuration document is internally represented by a configuration graph. In the example given in fig. 11, the configuration graph represents the evolution history of both simple documents and configurations. Furthermore, it contains information about the structure of configurations themselves, which consist of versions of interdependent documents.

In order to store versions efficiently, the graph deltas offered by GRAS may be used. Although

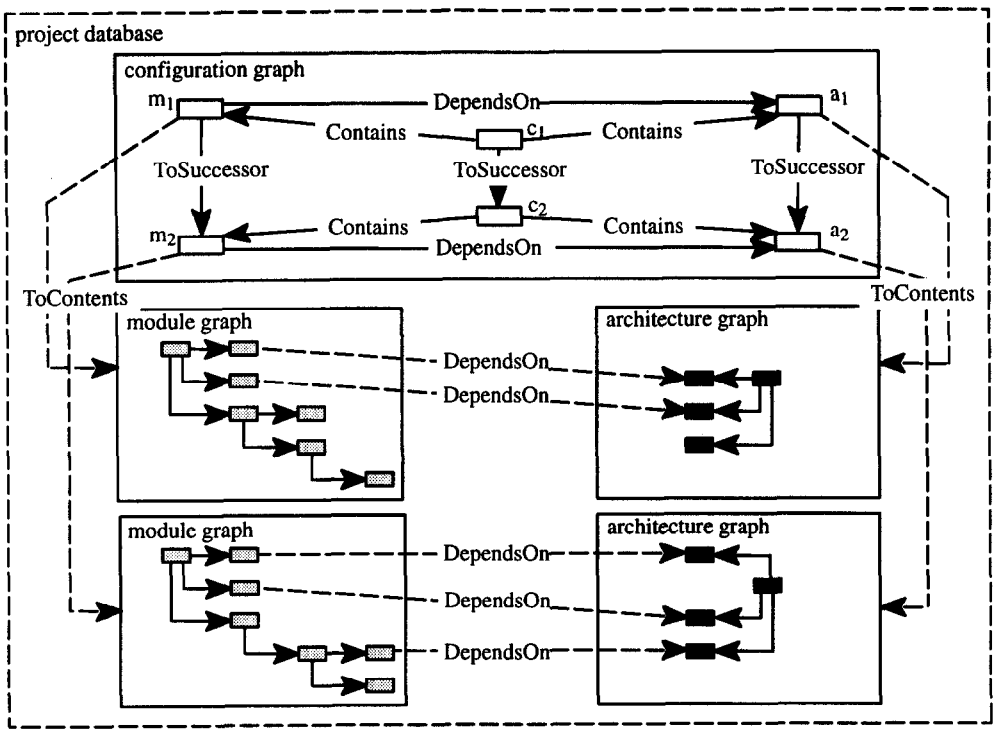


Fig. 11: Graph representation of a project database

GRAS does support version control on the physical level, it does not introduce a version model on the logical level because such a model should be defined by GRAS applications. In the example given in fig. 11, the version model is part of the configuration management model underlying configuration graphs (alternatively, the version model could have been specified separately).

Graphs for configurations and graphs for single documents differ radically with respect to their access patterns. This variety is supported by the *ClientServerDistribution* layer which is also responsible for concurrency control. Configuration graphs are typically accessed remotely in shared mode; locks are granted only during the execution of a single user command. Graphs representing single documents are normally accessed locally in exclusive mode (which avoids inter-process communication); locks are granted for a whole session.

Inter-document relations are represented by **inter-graph edges** (and by local edges contained in configuration graphs). Fig. 11 shows two kinds of them:

- fine-grained → coarse-grained: Each node representing a version of a simple document contains a reference to the corresponding document graph. These references represent the document hierarchy.
- fine-grained → fine-grained: Fine-grained relations between module implementations and the software architecture (e.g. dependencies between body and specification of an exported procedure) are represented by edges which connect nodes belonging to different graphs. Such edges are employed to support traceability (e.g. a procedure body may be traced back to its specification) and incremental change propagation [81] (e.g. the name of a procedure body is kept consistent with the name of its specification).

GRAS does not provide built-in support for representing inter-document relations. Multiple simulations have been developed on top of GRAS which differ with respect to efficiency, referential integrity, version control, concurrency control, and cardinality of edge types. A wide spectrum of simulations has been used in the IPSEN project, ranging from very simple to extremely complex ones (see e.g. [80]).

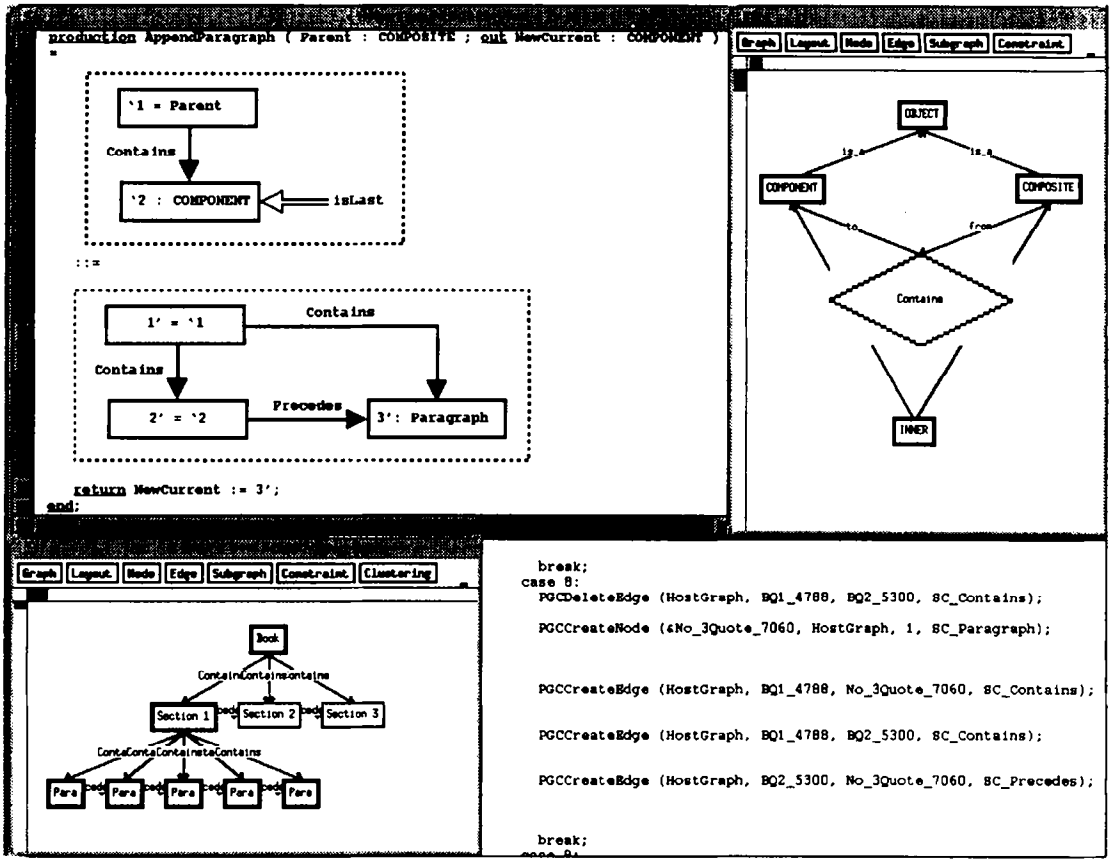


Fig. 12: Screen dump of the PROGRES database development environment

### *5.3. PROGRES development environment*

As already mentioned in previous sections, the GRAS system has a procedural interface for imperative programming languages like Modula-2 and C which merely supports insertion, deletion, and retrieval of single nodes, edges, and attribute values. Developing new applications on top of this interface is often tedious and error-prone. Therefore, we have designed PROGRES, a kind of **visual rule-oriented database programming language**, which supports retrieval and replacement of very complex subgraphs (in GRAS) as atomic operations [70, 71]. A complete formal definition of this language is based on a logic-oriented calculus of programmed graph rewriting systems [71, 72].

The realization of an **integrated database development environment** on top of GRAS based on the language PROGRES is nearby completion [62, 82, 83]. It consists of the following components:

- a syntax-aided editor for defining graph schemes and graph rewrite rules, which supports the manipulation of graphical as well as textual representations of PROGRES specifications,
- an incrementally working type-checker, which guarantees a specification's correctness with respect to the PROGRES language's static semantics,
- a graph scheme browser, which presents an ER-like view of a given node class hierarchy and which relies on a slightly extended and adapted version of the system EDGE of the University of Karlsruhe [63],
- an execution tool, which translates (incomplete) specifications into intermediate code (incrementally and on demand) for an abstract graph rewriting machine and interprets this code

afterwards,

- two compiler back-ends which translate the PROGRES intermediate code into pure Modula-2 and C code, respectively,
- and finally a graph browser, which is again based on EDGE and inspects graphs manipulated by the execution tool.

A first prototype of this environment comprising more than 400,000 lines of code is already available as free software.[†] The screen dump of fig. 12 provides the reader with an overall impression of its current status. There, we have

- one window (top/right) displaying a cut-out of our running example's graph scheme with four node classes and one edge type,
- one window (bottom/left) displaying an instance graph of a very small book which consists of one section with visible contents (five paragraphs) and two sections with invisible contents,
- one window (top/left) displaying the definition of a graph rewrite rule for appending a new *Paragraph* as the last child (parameter *At*) of a given *COMPOSITE* node (parameter *Parent*),
- and a cut-out (bottom/right) of the generated C code for this graph rewrite rule, i.e. that part of code which adds the new *Paragraph* node and connects this node to its neighbors by incoming *Precedes* and *Contains* edges.

## 6. RELATED WORK

The purpose of this section is to relate GRAS to other database systems for engineering applications. Note that some comparisons to related work have already been performed in the preceding sections. In the following, we will summarize and amplify these remarks.

### *6.1. Data model*

The GRAS data model relies on **attributed graphs** and supports classification by a stratified type system, specialization by multiple inheritance of node classes, aggregation by attributed nodes, association by binary, directed relations which do not carry attributes, and derived information by paths and derived attributes.

There are a few other database systems which also rely on attributed graphs, e.g. Neptune's Hypertext Abstract Machine HAM [19], Cactis [43], Adage [34], and HyperLog's storage manager [78]. Their data models agree with the GRAS data model inasmuch as objects are represented by attributed nodes and relations are modeled as edges which do not carry attributes. However, neither Cactis nor Adage have a stratified type system, and HyperLog's storage manager as well as Neptune's HAM even do not know the concept of "scheme consistent graphs".[‡] Furthermore, only Cactis supports derived information. However, Cactis does not provide for derived relations, and derived attributes are confined to the 1-context.

A current disadvantage of GRAS consists in its limited support of aggregation: In Adage and HyperLog's storage manager, graphs may be used as values of node attributes, resulting in hierarchical graphs. This feature is not yet supported by GRAS. However, integration of hierarchical graphs into the GRAS data model (and the GRAS system) is currently being investigated (see also section 5).

Finally, we have to mention two other graph-oriented database systems, namely Hy+ with its query language GraphLog [17], and GOOD [37]. Both systems are incomparable with GRAS

[†]Via anonymous ftp from: ftp.informatik.rwth-aachen.de in /pub/unix/PROGRES.

[‡]Telos [60] is an example of a data model supporting a stratified type system with an arbitrary number of type-instance levels (while GRAS only supports three levels). However, the data model differs from attributed graphs because it does not distinguish between attributes and edges.

inasmuch as they are visual front-ends to relational database systems or knowledge bases and not complete database systems in their own right. Therefore, they play about the same role for their back-ends as the PROGRES environment for GRAS, and it would even be interesting to use GRAS as a new and probably more efficient back end for these systems.

In addition to attributed graphs, other data models have been employed for software engineering database systems. **Abstract syntax trees** have been used in various syntax-aided software development environments (or generators for such environments) such as e.g. Gandalf [38] and CPSG [67]. Abstract syntax trees may be—and have actually been—simulated on top of GRAS. The advantage of using a more general framework—graphs instead of trees—lies in the uniform representation of tree and non-tree relations, i.e. abstract syntax trees may be augmented with edges representing control flow, data flow, etc. This leads to the more general notion of abstract syntax graphs [27].

Database systems such as DAMOKLES [23] and PCTE [28] are based on **extended Entity-Relationship models** (which ultimately have their roots in the ER model proposed by Chen [14]). Different variants of EER models exceed the GRAS data model by providing (a subset of) features like n-ary relationships, attribution of relationships, or predefined categories of relationships. For example, PCTE supports binary, attributed relationships which are classified into five predefined categories (*composition*, *reference*, *implicit*, *designation*, and *existence* in the PCTE ECMA standard). We have decided to avoid commitment to a complicated data model which incorporates many decisions which are subject to debate. Rather, we have kept our data model as simple as possible, and we offer the user a powerful data definition and manipulation language for specifying different variants of EER models as desired.

Recently, several database systems (e.g. O2 [20], Objectstore [51], and GemStone [11]) have been developed which are based on an **object-oriented data model**. In these systems, data manipulations are described in some sort of object-oriented programming language. As pointed out in section 5, the PROGRES environment realized on top of GRAS supports declarative rather than procedural specification of graph transformations. A disadvantage of GRAS/PROGRES consists in the fact that only evaluation rules for derived attributes can be inherited and redefined along the node class hierarchy. We are currently extending the PROGRES data model to provide inheritance of structural operations (graph tests and graph transformations) on the graph level. This extension is going to be integrated with the introduction of hierarchical graphs mentioned above.

The main advantage of the GRAS system in comparison to object-oriented database systems is that it directly supports the data model of directed graphs. Therefore, no additional efforts are necessary to implement a more or less sophisticated graph data model on top of an object-oriented database system's interface. Furthermore, the GRAS system's implementation is able to take care about the requirements of typical graph processing applications on all of its implementation layers as for instance:

- guaranteeing referential integrity for edges,
- supporting efficient bidirectional traversal of edges with (almost) arbitrary fan-out and fan-in, and
- keeping derived attributes and relations up-to-date.

It is still an open question whether a graph-oriented system such as GRAS may be implemented efficiently on top of an object-oriented database system. In addition to the problems mentioned above, the success of such an implementation effort heavily depends on the way objects are implemented. We are convinced that we need a distinction between heavy-weight objects (units of distribution and concurrency control) and light-weight objects in order to implement graphs and nodes, respectively. Implementing nodes as heavy-weight objects will be far too inefficient.

### *6.2. Architecture: the kernel*

The GRAS system's kernel consists of layers of data abstraction each of which has an own, carefully designed data model: the graph-oriented layer has been implemented on top of a record-

oriented layer which in turn relies on a page-oriented layer and a network file system layer. Furthermore, the central subsystem of the GRAS kernel, the *VirtualRecordStorage*, is highly parameterized and allows for the instantiation of rather different persistent data storages, as e.g. the attribute and edge storages of our graph-oriented layer. In this way, the GRAS system's implementation follows the so-called "tool kit" approach of systems like Exodus [12]. Therefore, it may be reused to realize persistent data storages for a wide spectrum of different applications.

The implemented indexing scheme of *VirtualRecordStorage* uses tries for addressing all relevant data pages within one storage, and static hashing with an order preserving function for locating relevant (ranges of) records within selected pages. Furthermore, a special record identifier creation algorithm (cf. [48]) guarantees clustering of logically related data and uniform distribution of records over pages. Thus, our indexing scheme is well-suited for dynamically growing and shrinking medium sized databases (with each database having the size of a typical engineering document). In this case, tries for addressing pages but not the addressed data itself fit into main memory. On the contrary,

- main memory indexing schemes—like those described in [3] and [54]—are tuned for the case that a whole database fits into main memory,
- whereas B-trees [7] and their variants assume that index structures themselves do no longer fit into main memory and that traversing these structures requires disk accesses, too,
- and dynamic hashing techniques [25] which either use directories—like 'extendible hashing' [30], 'virtual hashing' [56] etc.—or which address their data without any index at all—like 'linear hashing' [52], 'modified dynamic hashing' [45] etc.—suffer from the following drawback: they require global reorganizations of index structures or data from time to time in order to avoid expensive maintenance of overflow chains.†

### *6.3. Architecture: enhancing layers*

The event management layer, the first one on top of the GRAS kernel, monitors all basic graph state transitions and forwards them to a generic *EventTriggerActionMachine*. As already indicated by its name, the event mechanism provided by GRAS does not follow the lines of the wide-spread event-condition-action paradigm (cf. e.g. [57]), where an action is triggered whenever a certain event happens and a specified condition is fulfilled. In our opinion, it is sufficient to have more or less complex event patterns‡ and event triggered actions which may be composed of GRAS interface operations in an arbitrary manner. Events and actions may be used—as in other active database systems [8, 18, 24]—to control dynamic integrity constraints and to extend the functionality of already defined transactions, and they may even be used to integrate different applications on top of GRAS (in a similar way as in so-called broadcast/message server architectures [66]).

The change management layer extends the functionality of the kernel by providing undo/redo, nested transactions, and deltas. During a session (between *Open* and *Close*), the application can set an arbitrary number of checkpoints which define boundaries of top-level transactions and act as targets of undo/redo operations. Use of graph deltas is controlled by the application; furthermore, deltas are supported without introducing a logical model for version control. All services mentioned above are implemented in a uniform way by logs of graph operations. In order to keep deltas short, logs are maintained on a high level of abstraction (operations such as creation/deletion of nodes/edges rather than byte-oriented operations).

Comparing our approach to other work, we observe that traditional version control systems such as SCCS [68] or RCS [76] are different in the following respects: Applications cannot control the use of deltas, modeling of version control is mixed up with its realization, and deltas are

†Analysis and simulation results of [29] and [53] indicate that the above mentioned dynamic hashing techniques might be superior to our approach in the case of large databases, which consist of many thousands of pages and which would require deep tries.

‡Research is under way to allow client processes to define complex events which correspond to composite graph state transitions, as proposed in [32].

constructed a-posteriori[§] such that undo/redo has to be supported by a different mechanism. In contrast, more recent systems such as Gypsy [15] and EXODUS [12] are closer to GRAS inasmuch as they provide for flexible delta control and separate modeling from realization of version control.

While Gypsy (unlike GRAS) relies on a-posteriori construction of deltas, EXODUS constructs deltas on the side when performing change operations. In contrast to GRAS, EXODUS relies on data sharing: Data which are common to multiple versions are physically shared among them. This is achieved by applicative operations on tree-like structures (EXODUS uses B+ trees for storing file objects). Similar techniques have been applied e.g. in [31] and [1]. Although this approach seems attractive because versions are always immediately accessible,[†] it has been ruled out for the following reason: Data sharing techniques as they have been applied in other systems operate on a low level of abstraction. Within the GRAS system, such a technique would have to be applied on the page level: Each graph is realized by an index tree whose leaves point to storage pages. Small changes on the graph level (e.g. creation of a single node) may imply comprehensive physical reorganizations on the page level. Therefore, deltas on this level would become too large to be useful for an application. Note that this is not an argument against data sharing in general; rather, we argue against the use of data sharing on a low level of abstraction.[‡]

The scheme and attribute management layer is mainly responsible for computing intensionally defined facts in the form of derived attributes and relations. Table 1 summarizes the properties of our incremental evaluation algorithm in comparison to approaches used in a number of quite different systems:

- GOM [46, 57], an object-oriented database system, which supports the materialization of arbitrary functions and path expressions,
- Relational Attribute Grammars [40], RAG, which are an attempt to combine the tree-oriented data model of attribute grammars, found in CPSG [67], with the relational data model,
- and finally the already mentioned graph-based system Cactis [42, 43].

| | GOM | RAG | Cactis | GRAS |
|---|---|---|---|---|
| Lazy evaluation | + | – | + | + |
| Eager evaluation | + | + | – | – |
| Space efficiency | – | – | + | + |
| Incrementality | + | – | + | + |
| N–context dep. | + | – | – | + |

Table 1: Comparison of different incremental, derived data evaluation algorithms

All presented systems, with the exception of RAG, use a two-phase lazy evaluation algorithm for recomputing derived data. Furthermore, the system GOM offers a second eager evaluation algorithm which recomputes invalid data immediately after any changes to the underlying object base. In this way GOM avoids the invalidation of data that depends on potentially affected, but in fact still valid data. In rare cases, this algorithm may require exponential time because data are recomputed too early. Relational attribute grammars in contrast exploit very sophisticated attribute evaluation techniques to circumvent the above mentioned "exponential time complexity" trap. On the other hand, they have problems with mutual attribute/relation dependencies.

[§]An algorithm for a-posteriori construction of deltas which is applicable to arbitrary non-text files (in contrast to the built-in algorithms of SCCS and RCS) is presented in [65].

[†]When following the operation-based approach, reconstruction costs may be reduced by introducing a cache of recently reconstructed versions (as it is done e.g. in Gypsy).

[‡]The Smalltalk-based PIE system [35] which operates on the graph level and arranges nodes and attributes in different layers is not a satisfying solution since the runtime of operations increases with the number of layers which have to be searched (roughly speaking, for each version a new layer is needed which contains the changes relative to the previous version).

Furthermore, the propagation of changed information from relations to attributes is very coarse-grained. It suffers from the fact that even small modifications of relations lead to reevaluation of all attributes which depend on these relations.

Concerning space overhead, Cactis and GRAS, which both rely on static dependency graphs, are superior to GOM and RAG: the system GOM creates large dependency tables on the instance level, and the incremental relation maintenance algorithm of RAG relies on the construction of many intermediate relations. Comparing the ability of the four systems to maintain 'far-reaching' n-context dependent relations, the systems GOM and GRAS are equal to each other. The system Cactis is restricted to the 1-context case, and RAG seems to have problems to deal with transitive closure for arbitrary relations.

The concurrency and distribution layer extends the kernel by allowing different applications to share graphs in a secure and efficient way. In contrast to other distributed (object-oriented) database systems which exchange data (normally in page or object units) between clients and servers (e.g. [11] or [51]), GRAS exchanges operation calls and results. This concept has been successfully used e.g. in the X Window system [69]. Although data exchange on the physical or logical level as realized by page servers or object servers shows a better performance in general (because much of the actual work can be shifted from the server to the clients), this approach has been chosen because in most cases, a graph is used either by only one application exclusively or by many applications for frequent but short times:

- When many applications access a graph for short updates or queries, e.g. when operating on a project configuration graph, the lock protocols between clients and servers for ensuring consistency in the case of data exchanges are even more expensive than simple remote procedure calls.
- For achieving high performance of exclusively working applications as e.g. an analyzer checking a large graph, all kind of interprocess communication should be avoided at all, that is the database engine should better be linked directly to the application.

The layered architecture of GRAS allowed a straightforward realization of the direct linking of one application to a graph server. An implementation is currently under way that even performs an automatic and transparent shift from direct linking to network coupling when a graph is accessed by more than one application.

## 7. CONCLUSION

We have presented a graph-oriented database system which provides for maintenance of derived attributes and relations, undo/redo, nested transactions, deltas, event handling, and client/server distribution. Our presentation has covered data model, architecture, and applications of GRAS. Currently, a stable and efficiently working GRAS version with multiple-reader/single-writer concurrency control and with interfaces for the programming languages Modula-2 and C is available as free software.† Only recently, the implementation of the *ClientServerDistribution* layer has been included into this release. The current GRAS version is written in Modula-2, C, and C++; it comprises 130,000 lines of code. We are currently preparing a Modula-3 version of GRAS. Furthermore, we are extending GRAS in order to support inheritance of graph operations, hierarchical graphs, and inter-graph edges.

*Acknowledgements* — The development of the GRAS system has been supported by grants from the "German Research Council" (DFG Na 134/2-1, 2-2 and DFG Na 134/4-1, 4-2), the "Stiftung Volkswagenwerk" (VW I.61512), and the "European Community" (Esprit Project 5409). This research project was also made possible while one of the authors was working for the Research & Development group of GAK at Amsterdam (many thanks to Thiel Chang!). In addition to the authors, the following persons have contributed significantly to the design and implementation of the GRAS system: Roland Baumann, Thomas Brandes, Richard Breuer, Michael Broekmanns, Frank Höfer, Peter Klein, Claus Lewerentz, Wolfgang Reimesch, Ralf Spielmanns, and Stefan Zohren. Furthermore, we would like to thank the unknown reviewers for their constructive comments which in particular stimulated us to include a section on experiences and applications.

†Under the GNU license conditions of the "Free Software Foundation" via anonymous ftp from: ftp.informatik.rwth-aachen.de in `/pub/unix/GRAS<version_no>`.

## REFERENCES


[1] A. Alderson. A Space-Efficient Technique for Recording Versions of Data. *Software Engineering Journal*, IEE & BCS Joint Publ., 240-246 (1988).

[2] B. Alpern, A. Carle, B. Rosen. Incremental Evaluation of Attributed Graphs. Technical Report CS-87-29, Brown University (1987).

[3] A. Analyti, S. Pramanik. Fast Search in Main Memory Databases. In: [75], pp. 215-224.

[4] T.L. Anderson, A.J. Berre, M. Mallison et al.. The Hypermodel Benchmark. In: Bancilhon, Thanos, Tsichritzis (eds.): *Advances in Database Technology - EDBT '90*, LNCS 416, Springer Press, pp. 317-331 (1990).

[5] J.E. Archer, R. Conway, F.B. Schneider. User Recovery and Reversal in Interactive Systems. *ACM Transactions on Programming Languages and Systems*, vol. 6-1, ACM Press, pp. 1-19 (1984).

[6] F. Bancilhon, C. Delobel, P. Kanellakis (eds.). *Building an Object-Oriented Database System*. Morgan Kaufmann Publ. (1992).

[7] R. Bayer, E.M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, vol.1, Springer Press, pp. 173-189 (1972).

[8] C. Beeri, T. Milo. A Model for Active Object Oriented Database. *Proc. 17th Int. Conf. on Very Large Data Bases*, IEEE Computer Society Press, pp. 337-349 (1991).

[9] P. Bernstein. Database System Support for Software Engineering. *Proc. of the 9th Int. Conf. on Software Engineering*, IEEE Computer Society Press, pp. 166-178 (1987).

[10] T. Brandes, C. Lewerentz. GRAS : A Non-Standard Database System within a Software Development Environment. *Workshop for Software Engineering Environments for Programming in the Large*, Harwichport, Massachusetts, pp. 113-121 (1985).

[11] P. Butterworth, A. Otis, J. Stein. The GemStone Object Dabase Management System. *Communications of the ACM*, vol. 34-10, ACM Press, pp. 64-77 (1991).

[12] M. Carey, D.J. DeWitt, J.E. Richardson et al.. Storage Management for Objects in EXODUS. In: Kim, Lochovsky (eds.): *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Frontier Series, pp. 341-369 (1989).

[13] R.G.G. Cattell, J. Skeen. Object Operations Benchmark, *ACM Transactions on Database Systems*, vol. 17-1, ACM Press, pp. 1-31 (1992).

[14] P.P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, vol. 1-1, ACM Press, pp. 9-36 (1976).

[15] S.E. Cohen, D.A. Sari, R. Genecker et al.. Version Management in Gypsy. In: [39], pp. 201-215.

[16] J. Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, pp. 17-41 (1987).

[17] M. Consens, A. Mendelzon. Hy+: A Hypergraph-based Query and Visualization System. *Proc. ACM SIGMOD '93 International Conference on Management of Data*, SIGMOD Record, vol. 22-2, ACM Press, pp. 511-516 (1993).

[18] U. Dayal et al.. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD*, vol. 17-1, ACM Press, pp. 51-70 (1988).

[19] N. Delisle, M. Schwartz. Neptune: a Hypertext System for CAD Applications. *Proc. ACM SIGMOD '86 International Conference on Management of Data*, SIGMOD Record, vol. 15-2, ACM Press, pp. 132-143 (1986).

[20] O. Deux. The O2 System. *Communications of the ACM*, vol. 34-10, ACM Press, pp. 34-48 (1991).

[21] W. Deiters, V. Gruhn. Managing Software Processes in the Environment MELMAC. *Proc. of the 4th Int. Symposium on Practical Software Development Environments*, SIGSOFT Notes, vol. 15-6, ACM Press, pp. 193-205 (1990).

[22] D.J. DeWitt, D. Maier. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. *Proc. 16th Int. Conf. on Very Large Data Bases*, IEEE Computer Society Press, pp. 107-121 (1990).

[23] K.R. Dittrich, W. Gotthard, P.C. Lockemann. DAMOKLES, A Database System for Software Engineering Environments. *Proc. of the Int. Workshop on Advanced Programming Environments 1986*, LNCS 244, Springer Press, pp. 353-371 (1986).

[24] K.R. Dittrich, A.M. Kotz, J.A. Mülle. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. *ACM SIGMOD*, vol. 15-3, ACM Press, pp. 22-36 (1986).

[25] R. Enbody, H. Du. Dynamic Hashing Schemes. *ACM Computing Surveys*, vol. 20-2, ACM Press, pp. 85-113 (1988).

[26] G. Engels, U. Hohenstein, U. Hülsmann et al.. CADDY - Computer Aided Design of Non-Standard Databases. In: Madhavji, Schäfer, Weber (eds.): *Proc. of the 1st Int. Conf. on System Development Environments & Factories*, Pitman Press, pp. 151-158 (1989).

[27] G. Engels, C. Lewerentz, M. Nagl et al.. Building Integrated Software Development Environments Part I: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, vol 1-2, ACM Press, pp. 135-167 (1992).

[28] European Computer Manufacturers Association. *ECMA PCTE Standard*. ECMA-149 (1990).

[29] Ph. Flajolet. On the Performance Evaluation of the Extendible Hashing and Trie Searching. *Acta Informatica*, vol. 20, Springer Press, pp. 345-367 (1983).

[30] R. Fagin, J. Nievergelt, N. Pippinger et al.. Extendible Hashing: A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, vol. 4-3, ACM Press, pp. 315-344 (1979).

[31] C.W. Fraser, E.W. Myers. An Editor for Revision Control. *ACM Transactions on Programming Languages and Systems*, vol. 9-2, ACM Press, pp. 277-295 (1986).

[32] N.H. Gehani, H.V. Jagadish, O. Shmueli. Event Specification in an Active Object-Oriented Database. In: [75], pp. 81-90.

[33] C. Gerlhof, A. Kemper, Ch. Kilger et al.. Clustering in Object Bases. Technical Report 6/92, Fakultät für Informatik, University of Karlsruhe, Germany (1992).

[34] J. Giavitto, G. Rosuel, A. Devarenne, A. Mauboussin. Design Decisions for the Incremental Adage Framework. *Proc. of the 12th Int. Conference on Software Engineering*, IEEE Computer Society Press, pp. 86-95 (1990).

[35] I.P. Goldstein, D.G. Bobrow. A Layered Approach to Software Design. Technical Report, XEROX PARC, Palo Alto (1982).

[36] J. Gray (ed.). *The Benchmark Handbook*. Morgan Kaufmann Publ. (1991).

[37] M. Gyssens, J. Paraedaens, D. van Gucht. A Graph-Oriented Object Database Model for Database End-User Interfaces. *Proc. ACM SIGMOD Int. Conference on Management of Data*, SIGMOD Record, vol. 19-2, ACM Press, pp. 24-33 (1990).

[38] N. Habermann, D. Notkin. Gandalf : Software Development Environments. *IEEE Transactions on Software Engineering*, vol. 12-2, IEEE Computer Society Press, pp. 1117-1127 (1986).

[39] P. Henderson (ed.). *Proc. ACM 3rd Symposium on Practical Software Development Environments*. ACM SIGPLAN Notices, vol. 24-2, ACM Press (1989).

[40] S. Horwitz, T. Teitelbaum. Generation Editing Environments Based on Relations and Attributes. *ACM Transactions on Programming Languages and Systems*, vol. 8-4, ACM Press, pp. 577-608 (1986).

[41] S.E. Hudson. Incremental Attribute Evaluation: An Algorithm for Lazy Evaluation in Graphs. Technical Report 87-20, University of Arizona (1987).

[42] S.E. Hudson, R. King. The Cactis Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering*, vol. 14-6, IEEE Computer Society Press, pp. 709-719 (1988).

[43] S.E. Hudson, R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems*, vol. 14-3, ACM Press, pp. 291-321 (1989).

[44] Special Issue on the Interface Description Language IDL. *ACM SIGPLAN Notices*, vol. 22-11, ACM Press (1987).

[45] K. Kawagoe. Modified Dynamic Hashing. In: Navathe (ed.): *Proc. of the ACM SIGMOD 1985 Int. Conf. on Management of Data*, ACM Press, pp. 201-213 (1985).

[46] A. Kemper, G. Moerkotte. Access Support Relations: An Indexing Method for Object Bases. *Information Systems*, vol. 17-2, Pergamon Press, pp. 117-145 (1992).

[47] U. Keßler, P. Dadam. Evaluation of Complex Queries on Hierarchically Structured Objects by Path Indexes. In: Appelrath (ed.): *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft*, IFB 270, Springer Press, pp. 218-237 (1992).

[48] N. Kiesel, A. Schürr, B. Westfechtel. Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications. Technical Report AIB 92-44, Technical University of Aachen, Germany (1992).

[49] N. Kiesel, A. Schürr, B. Westfechtel. GRAS, a Graph-Oriented Database System for (Software) Engineering Applications. *Proc. 6th Int. Workshop on Computer-Aided Software Engineering*, IEEE Computer Society Press, pp. 272-286 (1993).

[50] M.F. Kilian. A Note on Type Composition and Reusability. *OOPS Messenger*, vol. 2-3, ACM Press, pp. 24-32 (1991).

[51] O. Lamb, G. Landis, J. Orenstein et al.. The Objectstore Database System. *Communications of the ACM*, vol. 34-10, ACM Press, pp. 50-63 (1991).

[52] P.A. Larson. Linear hashing with Parital Expansions. *Proc. of the 6th Int. Conf. on Very Large Databases*, IEEE Computer Society Press, pp. 224-232 (1980).

[53] P.A. Larson. Dynamic Hash Tables. *Communications of the ACM*, vol. 31-4, ACM Press, pp. 446-457 (1988).

[54] T.J. Lehmann, M. Carey. A Study of Index Structures for Main Memory Database Management Systems. *Proc. of the 12th Int. Conf. on Very Large Databases*, IEEE Computer Society Press, pp. 294-303 (1986).

[55] C. Lewerentz, A. Schürr. GRAS, a Management System for Graph-Like Documents. In: Beeri, Schmidt, Dayal (eds.): *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases*, Morgan Kaufmann Publ., pp. 19-31 (1988).

[56] W. Litwin. Virtual Hashing: Dynamically Changing Hashing. *Proc.of the 4th Conf. on Very Large Databases*, IEEE Computer Society Press, pp. 517-523 (1978).

[57] D.R. McCarthy, U. Dayal. The Architecture of An Active Database Management System. In: Clifford, Lindsay, Maier (eds.): *Proc. of the ACM SIGMOD 1989 Int. Conf. on Management of Data*, ACM Press, pp. 215-224 (1989).

[58] A.R. Meyer, M.B. Reinhold. 'Type' is not a type. In: *Proc. of the 13th ACM Symp. on Principles of Programming Languages*, ACM Press, pp. 287-295 (1986).

[59] H.A. Müller, K. Klashinsky. Rigi - A System for Programming-in-the-large. *Proc. 10th Int. Conf. on Software Engineering*, IEEE Computer Society Press, pp. 80-85 (1988).

[60] J. Mylopoulos, A. Borgida, M. Jarke et al.. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, vol. 8-4, ACM Press, pp. 325-362 (1990).

[61] M. Nagl. Characterization of the IPSEN Project. In: Madhavji, Schäfer, Weber (eds.): *Proc. of the 1st Int. Conf. on Systems Development Environments & Factories 1989*, Pitman Press, pp. 141-150 (1990).

[62] M. Nagl, A. Schürr. 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 Press, pp. 599-609 (1991).

[63] F. Newbery Paulisch. The Design of an Extendible Graph Editor. Dissertation, University of Karlsruhe, Germany (1991).

[64] B. Peuschel, W. Schäfer. Concepts and Implementation of a Rule-Based Process Engine. *Proc. of the 14th Int. Conf. on Software Engineering*, IEEE Computer Society Press, pp. 262-279 (1992).

[65] C. Reichenberger. Delta Storage for Arbitrary Non-Text Files. *Proc. of the 3rd Int. Workshop on Software Configuration Management 1991*, ACM Press, pp. 144-152 (1991).

[66] St. Reiss. Interacting with the FIELD Environment. *Software: Practice and Experience*, vol. 20-S1, John Wiley & Sons, pp. 89-115 (1985).

[67] T. Reps, T. Teitelbaum. *The Synthesizer Generator*. Springer Press (1988).

[68] M.J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, vol. 1-4, IEEE Computer Society Press, pp. 364-370 (1975).

[69] R. Scheifler, J. Gettys. The X Window System. *ACM Transactions on Graphics*, vol. 5-2, ACM Press, pp. 79-109 (1986).

[70] A. Schürr. 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 Press, pp. 641-659 (1991).

[71] A. Schürr. *Operational Specification with Programmed Graph Rewriting Systems: Formal Definitions, Applications, and Tools* (in German). Deutscher Universitäts Verlag (1991).

[72] A. Schürr. Logic Based Structure Rewriting Systems. *Proc. Dagstuhl Seminar 9301 on Graph Transformations in Computer Science*, LNCS 776, Springer Press, pp. 341-357 (1993).

[73] A. Schürr, A. Zündorf. Non-Deterministic Control Structures for Graph Rewriting Systems. In: Schmidt, Berghammer (eds.): *Proc. of the WG '91 Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 570, Springer Press, pp. 48-62 (1991).

[74] K.E. Smith, St.B. Zdonik. Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems. *Proc. of OOPSLA '87, Object-Oriented Programming Systemes, Languages and Applications*, SIGPLAN Notices, vol. 22-12, ACM Press, pp. 452-465 (1987).

[75] M. Stonebraker (ed.). *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, vol. 21-2, ACM Press (1992).

[76] W.F. Tichy. RCS - A System for Version Control. *Software: Practice and Experience*, vol. 15-7, John Wiley & Sons, pp. 637-654 (1985).

[77] M.M. Tsangaris, J.F. Naughton. On the Performance of Object Clustering Techniques. In: [75], pp. 144-153.

[78] E. Tuv, A. Poulovassilis, M. Levene. A Storage Manager for the Hypernode Model. *Proc. BNCOD-10 10th British National Conf. on Databases*, Aberdeen, Scotland, pp. 59-77 (1992).

[79] B. Westfechtel. Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control. Technical Report AIB 89-8, Technical University of Aachen, Germany (1989).

[80] B. Westfechtel. Revision Control in an Integrated Software Development Environment. *Proc. of the 2nd Int. Workshop on Software Configuration Management*, ACM SIGSOFT Software Engineering Notes, vol. 14-7, pp. 96-105 (1989).

[81] B. Westfechtel. A Graph-Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents. *Proc. of the 5th Int. Workshop on Computer-Aided Software Engineering*, IEEE Computer Society Press, pp. 2-13 (1992).

[82] A. Zündorf. Implementation of the Imperative/Rule Based Language PROGRES. Technical Report AIB 92-38, Technical University of Aachen, Germany (1993).

[83] A. Zündorf. Heuristic Solution for the (Sub-)Graph Isomorphism Problem in Executing PROGRES. Technical Report AIB 93-5, Technical University of Aachen, Germany (1993).