Object-oriented Database Management Systems for Construction of CASE Environments*

Wolfgang Emmerich, Petr Kroha and Wilhelm Schäfer

University of Dortmund, Dept. of Computer Science
D-44221 Dortmund, Germany
{emmerich|kroha|wilhelm}@ls10.informatik.uni-dortmund.de

Abstract. We argue that a fully object-oriented database management system is a very suitable basis of every modern CASE environment. We describe how the features provided by an OODBMS are exploited to build a CASE tool or environment. We discuss especially problems concerning inter-document consistency constraints and multi-user support. We finally sketch the features which are still missing in OODBMSs.

1 Introduction

The enormous increase in the size of software and the reliability required from modern software intensive systems has focussed attention on languages and corresponding tools which do not only support programming but also specification, design, and documentation of software. In fact, specification or design documents have become equally important as the final code in order to check and verify correctness, completeness and reliability of a delivered software product. In addition, the industrial-scale production of software today requires teams of developers who should be supported by tools which organise access to a large amount of shared and frequently changing information. This information consists of the above mentioned documents for the various phases of the software production process. Examples for such documents are data-flow diagrams, state-transition diagrams, petri-nets, entity-relationship diagrams, and modular design descriptions.

Tools supporting the construction of documents usually work in a syntax-directed mode, which is the adequate support for those mostly graphical languages. In more detail, the user selects a graphical element from a panel, places it on a drawing board and can possibly annotate it with textual explanations. This mode especially enables the tool to check the correct syntactic construction during editing.

Even for textual input, syntax-directed manipulation is useful, because it avoids a lot of typing errors (templates for the concrete syntax of a language are generated automatically) and, more than that, static semantic consistency

* This work has been partly funded by the CEC under contract No. 6115 (ESPRIT-III project GOODSTEP)

can immediately be checked, e.g. the use of a variable is checked against its declaration and the user is immediately informed about errors.

In general, syntax-directed tools support the syntactically and static semantically correct construction of documents which is often denoted as *intra-document consistency*.

An integrated software development or CASE-environment is a collection of tools supporting various phases of the production process. It has two main additional features compared to tools supporting the construction of single documents in a single language.

The first feature is the possibility to handle *inter-document consistency*, e.g. a name of a function or a parameter should be the same in the requirements specification, in the interface definition of a module, in the design document and in the implementation. Even more, changing the name in one document can result in a propagation of this change into the other documents concerned. If an environment maintains such inter-document dependencies, it is able to support an incremental intertwined development and maintenance of the documents. As a further illustration for the advantage of an incremental production process assume that a programmer may have detected an error in the code which is due to a wrong requirements specification of a particular function. If then the specification is changed, the environment could inform the user about all other places in all other documents which are affected by this change. In contrast, doing such a small incremental change in a phase-oriented environment is rather tedious, because such an environment is usually based on a complete transformation of all documents concerned from one phase into the next one. For more details we refer to [8].

The second main feature of an environment is *multi-user support* which means document change in general and in particular the above sketched change propagation must be subject to concurrency control and access rights defined for a team of developers, e.g. an automatic immediate update of one document as a consequence of a change in another document can not be performed in any case.

The two features of an environment, i.e. maintaining inter-document dependencies and even inter-document consistency and providing multi-user support demand complex-structured, persistent data and thus the use of a sophisticated database management system as a key architectural component of a CASE-environment. In commercial CASE-environment development this demand is often not (yet) taken very seriously, i.e. a lot of existing tools or environments are still based on rather rudimentary extensions of file systems. This is, to our view, due to the lack of appropriate database management systems for CASE (cf. Section 5 and and [7]).

This paper argues that fully object-oriented database management systems (OODBMSs) like O_2 or GEMSTONE are the most appropriate ones and it will illustrate how an integrated CASE environment is built on top of such a database system, i.e. it will explain the construction of the database scheme, and the exploitation of other database features like transaction management, and the performance capabilities offered by OODBMs. Thus, the next section will sketch the concept of an integrated CASE environment, whereas section 3 describes the scheme construction in terms of a class hierarchy of an object-oriented data definition and manipulation language. Section 4 explains how OODBMs have to be extended in order to fully meet the requirements of CASE environments. Section 5 sketches related work. Section 6 reports about the implementations we used for evaluating the results described in this paper.

2 The Concept of Integration in a CASE-Environment

Syntax-directed document manipulation and maintenance of inter-document consistency is based conceptually on a graph-like representation scheme of a document. The graph (usually called abstract syntax graph) describes the syntactic structure of each document [8, 1]). Additional edges describe inter-document dependencies. Operations which are performed by the user of a CASE-environment, are conceptually graph operations. They have to be defined in a way that they respect static semantics and inter-document consistency. Note, that consistency is given by the definition of those operations. For example, an operation which changes the name of a function in a design document could be defined in such a way that it performs the corresponding name change in all other documents concerned by traversing the graph along the edges connecting various occurrences of a function name in different documents.

As an example for a graph scheme (without considering the definition of the operations) see Fig. 1. It sketches the dependencies within and between three documents which are technical documentation, modular design and implementation of the module bodies. The solid arrows represent the syntactic structure of each document, i.e. the so-called abstract syntax tree. This tree is usually turned into a graph by indicating inter- and intra-document dependencies also by (dashed) edges. Finally, node attributes describe values like e.g. names of identifiers, modules etc.

More generally speaking, a project-wide ASG is a directed attributed graph which conceptually consists of a subgraph for each document. Each document subgraph in turn is spanned by a tree which is determined by the grammar of the language, the document is written in. Subtrees of this spanning-tree that are units for manipulation at the user-interface are called *increments*. An example of an increment of the design document depicted above is the procedure `InitWindowManager` with its identifier and parameter list. Edges of this spanning-tree are called *syntactic edges*. All other edges which consequently describe inter- and intra-document dependencies between nodes are called *non-syntactic edges*.

For the scope of this paper the given informal explanation of ASGs should do. A few languages have been developed (e.g. PROGRESS developed in the IPSEN project [8]) which allow to formally specify such rather complex graph schemes and especially the consistency preserving operations on those schemes.

The whole concept of integration is becoming more complicated if different users work on different documents at the same time. Then the execution of a particular operation on the graph by a particular user is subject to an access right which was granted to this user and allows the execution of the operation.

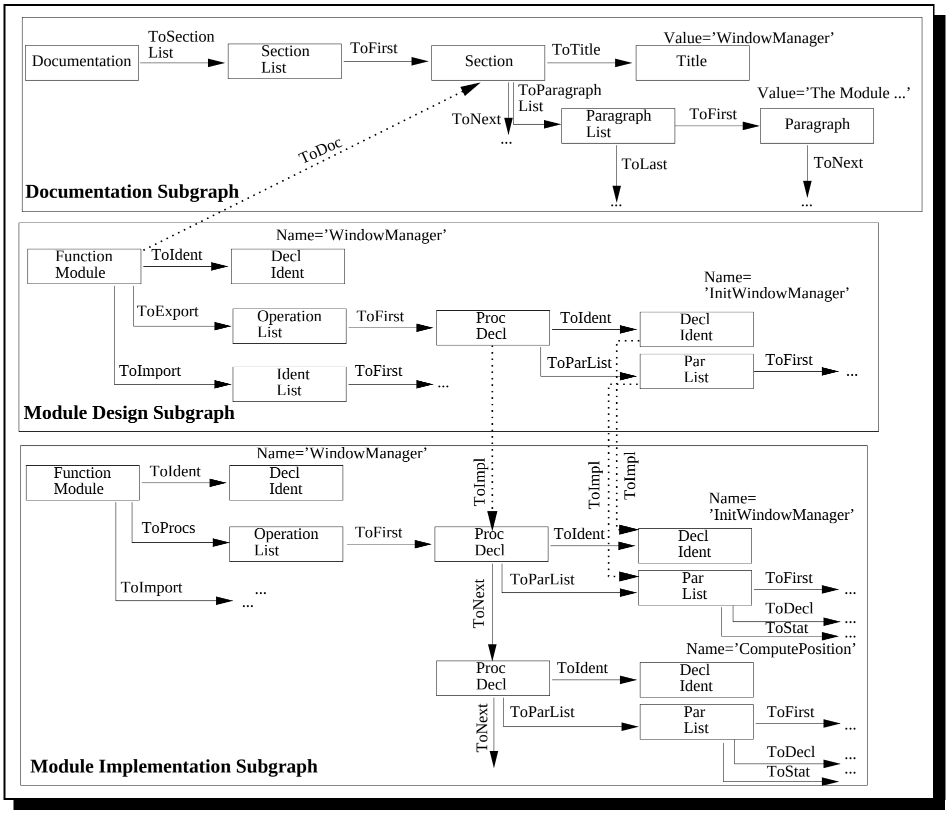


Fig. 1. Intra- and inter-document consistency

In addition, automatic change propagation across document boundaries have to be treated differently depending on the particular document, the owner of the document, and the state of the document. For example, if one user works on some specification and another one started already to develop a corresponding implementation, then change propagation should only happen after both have finished a major piece of their work and explicitly require the environment to renew document consistency (maybe only partly automatically). Thus the environment just has to keep track of inconsistencies for some time and it may only display warnings to users about possible inconsistent states.

In more sophisticated environments which however only exist as research prototypes so far, the concurrent manipulation of (integrated) documents is supported by versioning, i.e. users may change different versions of the same document concurrently. Then strategies for handling merging versions have to be defined as well. In addition, versioning is, of course, already a useful concept even in the single user case.

Finally, document representation and integration schemes as described undergo changes even during the construction of documents according to the defined scheme. For example, a user may add new document dependencies which have not been anticipated or the syntax definition of a language could change.

3 OODBMSs in a Multi-User CASE Environment

3.1 Scheme Definition and Generation

The objects stored in a database of a CASE environment represent user's documents, and, as we have argued, they have to be stored persistently in a structured way according to their syntax.
Consequently, a database scheme for a CASE environment firstly defines all possible syntactical constructions as classes according to the language definitions. The scheme defines additional classes for representing objects describing static semantics, e.g. a symbol table. Further a scheme defines syntactic and non-syntactic relationships among classes.

The overall integrity constraint of a database which all tools must obey is that document subgraphs must represent syntactically correct documents in which static semantics and inter-document consistency constraints are only violated in a controlled way. To enforce this constraint, we implement the constraints within the database scheme and exploit the fact that each tool (as a database application) can only perform those modifications that are in-line with the scheme defined.

We now present how we define the structure of abstract syntax graphs using an object-oriented scheme definition language. The common properties of nodes are defined within *classes* of the database scheme. Nodes are *complex objects* whose instance variables represent edges. Navigation along these edges is done by dereferencing instance variables. In case the number of edges that may start at a node is not known in advance, object constructors such as lists or sets are used. For navigation purposes a *query language* is used then.

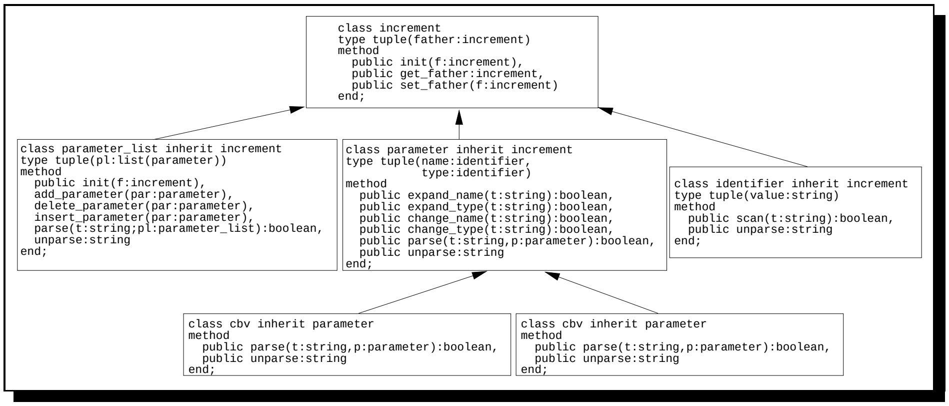
The type-compatibility in the scheme should be checked at compile-time in order to achieve safety and better performance. The set of target nodes of a particular edge should therefore be restricted to those types of nodes which are allowed according to syntax and static semantics of the language. Therefore, we exploit the *type-system* provided by typed OODBMSs (such as O_2) to define the types of instance variables.

For further scheme simplification, *inheritance* is used to define common properties of nodes such as outgoing syntactic or non-syntactic edges or attributes in a superclass only once. In addition, edges connect not always nodes of the same type. In case of alternative productions in grammars such as `A::=B|C`, we have edges that may connect different types of nodes. We must therefore allow that all edges which point to nodes of type A can also point to nodes of type B or C. In the example, we declare the classes which represent node types B and C to inherit from the class derived from A. Exploiting *polymorphism*, we can then assign instance variables of class A also values of class B and C.

Integrity constraints are enforced by *encapsulation*, i.e. applications are not allowed to modify instance variables directly, but must use the methods defined.

For example, terminal increment classes have a `scan` method which guarantees that values assigned to lexem attributes obey the lexical syntax. In case of the non-syntactic edge, we can declare a method with which a declaring identifier can be changed. That method may additionally perform a propagation of the change along all non-syntactic edges to all objects that represent the identifiers' usage. In the multi-user case, this method can additionally test whether the identifiers are contained in documents which the user is allowed to modify or not. Depending on that decision the method performs a change propagation or marks the identifier as inconsistent. Hence, we define multi-user support statically already in the scheme.

The computations necessary for performing these methods, however, require *computational completeness* of the scheme definition language.



```
class increment
type tuple(father:increment)
method
  public init(f:increment),
  public get_father:increment,
  public set_father(f:increment)
end;
```

```
class parameter_list inherit increment
type tuple(pl:list(parameter))
method
  public init(f:increment),
  add_parameter(par:parameter),
  delete_parameter(par:parameter),
  insert_parameter(par:parameter),
  parse(t:string;pl:parameter_list):boolean,
  unparse:string
end;
```

```
class parameter inherit increment
type tuple(name:identifier,
           type:identifier)
method
  public expand_name(t:string):boolean,
  public expand_type(t:string):boolean,
  public change_name(t:string):boolean,
  public change_type(t:string):boolean,
  public parse(t:string,p:parameter):boolean,
  public unparse:string
end;
```

```
class identifier inherit increment
type tuple(value:string)
method
  public scan(t:string):boolean,
  public unparse:string
end;
```

```
class cbv inherit parameter
method
  public parse(t:string,p:parameter):boolean,
  public unparse:string
end;
```

```
class cbv inherit parameter
method
  public parse(t:string,p:parameter):boolean,
  public unparse:string
end;
```

Fig. 2. Example of a Scheme Definition

A major advantage of scheme construction for CASE environments is that normalised grammars [8] of the languages can be used for deriving the scheme partly automatically. Each terminal and non-terminal symbol of the grammar is translated into a class. An instance variable of type string (for textual languages) used to store the value of lexems is attached to each terminal class. Instance variables for storing syntactic edges are attached to each non-terminal class according to the production on which the respective non-terminal appears on the left-hand side. The types of these instance variables are defined as the classes derived from the symbols on the productions' right-hand side. Classes which represent optional increments are declared to inherit from a predefined class representing the properties of optional increments. Symbols that appear on the right-hand side of an alternative production are transformed into subclasses of the class representing the symbol on the left-hand side. Productions like {`A`} are transformed into a class with an instance variable that is a list constructed from the class derived from the repeated symbol.

Methods for scanning lexems are generated from regular expressions and are

provided by each terminal class. Methods for parsing and construction of the spanning-tree which represents an increment are generated for each non-terminal class using parser-generation techniques. Methods for unparsing (i.e. to compute the external representation of data structures) are attached to all classes. Also, methods for expanding and collapsing increments which are frequently used in template-based editors can be attached to each non-terminal class.
The algorithm sketched is the basis of our scheme generator. So far, the definition of static semantics, inter-document consistencies and multi-user support has to be added manually by defining the respective methods.

Fig. 2 depicts an O_2 schema for parameter lists in PASCAL. Instance variables and method heads are depicted within solid boxes. Arrows denote inheritance.

3.2 Transaction Management

When users start a CASE tool, they start a session (as part of a long-durating transaction), that lasts until they quit the tool. Such a session can not be performed within one conventional database transaction (with ACID properties) for three reasons: Firstly, other users would not be able to use intermediate results produced within the session. Secondly, concurrency control conflicts would cause that other users would not be able to access those parts of their documents which have incoming or outgoing non-syntactic edges to the documents edited in the session. Finally, there is a likely chance to loose significant human effort in case of hard- or software failures.

Instead, we split a session into a number of short conventional transactions with ACID properties each of which executes the computations caused by a short user interaction. For example, changing a name of a type would be such an interaction which modifies the lexem attribute and all attributes of objects that represent identifiers that use the type. This strategy achieves that firstly, other users immediately see the effect of an interaction after transaction commit. Secondly, concurrency control conflicts become fairly rare, because they occur only if two sessions access the same nodes within two concurrent short transactions. Finally, there is no danger of loosing significant effort.

To implement the requirement of distributed access of users to documents, we can not execute tools on the machine on which the projects' database is stored. We would overload the server, since tools opposed to standard applications perform significant computation in order to create textual or graphical representations of the ASG, to compute context-sensitive menus and to display them on the screen. Instead we can exploit the client/server architecture offered by most OODBMSs in order to achieve process distribution.

3.3 Performance Capabilities

OODBMSs can only be used, if they achieve reasonable performance. To investigate this, we defined a software engineering application specific benchmark in order to evaluate the performance of OODBMSs. The Opus-Benchmark [6] accesses and modifies an ASG composed of several hundred document subgraphs with a high number of non-syntactic edges in between them. The benchmark simulates template based editing operations such as insertion, modification and deletion of increments as well as analysis operations which require massive traversals through the graph. The application of this benchmark to a few systems in particular an archetypical OODBMS like GemStone justifies the statement that those systems perform reasonably well with respect to software engineering applications. The description of the GemStone implementation as well as a detailed discussion of the results is out of the scope of this paper. Instead we only sketch the main results and refer to [5] for a detailed discussion.

With respect to space, storing ASGs in GemStone for instance is excellent. It required only about 2.5 times the space than storing a textual representation in a file system.

With respect to time, the response-time of tools increases with the number of transactions executed per time unit. In single-user case (i.e. no concurrent transactions at all) we observed that template-based editing operations perform in less than 100 milliseconds[2]. Unparsing medium-sized documents (up to 500 nodes) takes less than 500 milliseconds. Committing an optimistic transaction requires about 500 milliseconds. If incremental unparsing is chosen (i.e. only those parts of the textual document representations are redisplayed which have changed), interactions can be executed in about half a second and users are not going to recognise them as delays.

In multi-user case, we observed that response times become unacceptably worse, if more than four users work intensively (i.e. without significant thinking periods between successive interactions) on the same project-wide ASG.

4 Necessary Extensions of object-oriented DBMSs

4.1 Transaction Management

A major reason for the bad performance in multi-user mode is that the database spends unnecessary effort on achieving isolation of transactions. This is illustrated now.

One major paradigm in software engineering is information hiding. This means that users designing and implementing a large software system divide it into small modules with well defined small interfaces. As an example consider class definitions in C++ shown to other users while method implementations are hidden. Conceptually, those parts that represent hidden documents or fragments thereof do not have any non-syntactic edges to nodes of other documents. Usually, only one user modifies a document at a time. Therefore concurrency control conflicts can only be caused by transactions which access nodes along such non-syntactic edges (in order to check or preserve inter-document consistency).

[2] The times have been captured on a Sun SparcStation II with a medium-sized local SCSI disk.

Hence, in many transactions, there is no need for the DBMS to perform concurrency control. Tools can tell the DBMS at transaction start whether or not concurrency control is required. Note, that it is inappropriate to define this on session level, as both transactions with and without concurrency control may have to be executed.

Less time is needed if concurrency control is abandoned, because locking of objects or maintenance of conflict sets with conflict detection at commit-time need not be done. Note, that other transaction properties such as atomicity and durability must still be supported.

4.2 Version Management

Versioning of documents implies versioning of the corresponding subgraphs. As a prerequisite to have the database management system maintaining versions and revisions of subgraphs, the scheme definition language must offer means to define the notion of subgraphs. This can either be done at scheme generation time or at run-time. In the first case composite instance variables which lead to component objects are distinguished from non-composite instance variables which refer to objects [11]. We would then declare each instance variable which implements a syntactic edge as composite instance variable while non-syntactic edges are implemented as non-composite instance variables. In the second case, objects are added to a container that implements the composite object. Note, that in both cases the requirement that in a composite object, a dependent object is a component of only one composite object holds, because syntactic edges define a spanning-tree in the subgraph. We have then managed to implement document representing subgraphs as composite objects. The first solution sketched must be supported by the scheme definition language (as in the ORION system) whereas the second solution can be added as a general class for composite objects without modifying the OODBMS.

The operations which must be provided for versioning composite objects [12] must offer transparent versioning (i.e. to establish a current version), derivation of new versions, merging of alternate versions or retrieval of a particular version.

4.3 Scheme Evolution

In order to achieve changes of the syntax of documents, their static semantics and changes of inter-document consistency constraints during an ongoing software production process, the database must be able to perform incremental scheme updates. As an example consider the definition of a new reviewing strategy for module interfaces, which requires that documents are now annotated by the reviewers' name and have an additional relationship to a new document type "review report“.

Implementing changes of documents' structure, requires to add, rename or delete classes, to change the inheritance relation between classes, to add instance variables, to change their names and types, and to delete instance variables, and finally to create, change and delete methods. To cope with the above mentioned example, we have to add new classes which define the structure of the review report and to add new instance variables for storing reviewers names and the relationship to the review report to the class which represents module interfaces.

In order to preserve the integrity of existing documents the objects of the database must migrate to the new scheme. In the above example, module interface definitions must neither be deleted, nor manually be transformed to the new scheme.

The scheme evolution facilities available in current databases, however, do not fully cover those requirements. In GemStone for instance, objects are not affected by a scheme update, i.e. an object can only be accessed with the scheme that was established when the object was created. In O_2, objects must be manually transformed to conform to the new scheme.

5 Related Work

A major piece of work in CASE tool construction during the last ten years focussed on tool generators (e.g. [15, 1] which are similar to compiler-compilers in compiler construction. Those approaches do not consider to use a database system at all. All information produced during a working session with such a tool is just dumped into a file after the end of a session. Those approaches provide no multi-user support. In addition, inter-document consistency is also a problem, because the generators only work for a particular language and corresponding single documents and not for the definition and manipulation of document dependencies, i.e. project-wide abstract syntax graphs.

A few research projects in environment construction have built their own dedicated database systems like GRAS [13]. Those approaches, however, focus on an adequate persistent graph representation of the abstract syntax graphs. Thus, they enable a quick manipulation of arbitrary large graphs by smart caching techniques. They do not adequately support multi-user access. As a first step towards more sophisticated support, GRAS has been recently extended to deal with version management.

Some of the available commercial CASE tools or environments respectively use relational databases. They end up with the well-known performance problem of relational technology when being used for storing highly complex objects as abstract syntax graphs [14].

More recent research work has focussed on building dedicated software engineering database systems like PCTE/OMS [9] or DAMOKLES [4]. Unfortunately, these systems are only strong in efficiently supporting coarse-grained dependencies between documents. They do not adequately support the efficient manipulation of such fine-grained information.

In general, none of the mentioned systems offers a fully object-oriented data definition and manipulation language and thus lack the adequate modelling power for describing these complex structured software engineering data.

6 Implemented Systems and Further Work

We started using object-oriented DBMSs in an experimental evaluation of their performance. In [3] we describe a simple OMS benchmark we implemented on top of several structurally object-oriented databases (such as PCTE/OMS, Damokles, GRAS and Cadlab/OMS) as well as on GemStone and VBASE. It turned out that GemStone performed very well with small grained objects. Based on the results of that benchmark, we implemented the Opus benchmark [6] for those systems that performed well with fine-grained objects. According to the results we obtained, we selected GemStone for our further developments. We then ported the commercially available OPUS environment which supports design and implementation phase from GRAS to GemStone [10]. Thus, we were able to change OPUS from a single-user system to a multi-user system which is called Groupie. Using this environment, we also experienced the limits concerning transaction throughput mentioned in section 3.3. Furthermore, we implemented a generator which automatically derives a set of C++ classes that define the common properties of increments from a grammar written in a normalised BNF [2]. The C++ classes are generated in the way sketched in Section 3.1. These classes are then registered by GemStone's C++ Interface to become a part of the scheme definition.

We have just started implementing a generator which takes conceptual specifications of the syntax, static semantics, inter-document consistency and multi-user support and generates a database scheme. This work is a part of our activity in the ESPRIT-III project GOODSTEP (General Object-Oriented Database for SofTware Engineering Processes). In this project, O_2 will be enhanced in a way that it overcomes the deficiencies identified in section 4.

Acknowledgements

We are grateful to all members of the GoodStep consortium for intensive discussion about OODBMSs in CASE environments. In particular, we appreciated the discussions with Prof. C. Ghezzi and Prof. A. Fugetta about basic transaction mechanisms required from an OODBMS. Initial ideas regarding versioning evolved during discussions with Dr. J. Madec, Prof. J. Welsh and Prof. C. Delobel. Dr. S. Even, Ms. S. Sachweh, Prof. R. Zicari and Dr. R. de By provided us with deep insights in the topics of scheme updates and type safety.

We enjoyed working with a number of students working on this subject. F. Buddrus did a great job when implementing the first scheme generator. When implementing Groupie, M. Kampmann showed what is feasible and discovered the limits in the use of current OODBMSs.

References

1. P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

2. F. Buddrus. Generierung von syntaxgesteuerten Werkzeugen auf der Basis eines objektorientierten Datenbanksystems. Master's thesis, University of Dortmund, Dept. of Computer Science, June 1992.
3. S. Dewal, W. Emmerich, and K. Lichtinghagen. A Decision Support Method for the Selection of OMSs. In *Proc. of the* 2^{nd} *Int. Conf. on Systems Integration, Morristown, N.J.*, pages 32–40. IEEE Computer Society Press, 1992.
4. K. R. Dittrich, W. Gotthard, and P. C. Lockemann. Damokles – a database system for software engineering environments. In R. Conradi, T. M. Didriksen, and D. H. Wanvik, editors, *Proc. of an Int. Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 353–371. Springer, 1986.
5. W. Emmerich and M. Kampmann. The Merlin OMS Benchmark – Definition, Implementations and Results. Technical Report 65, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1992.
6. W. Emmerich and W. Schäfer. Dedicated Object Management Benchmarks for Software Engineering Applications. In R. Welland, editor, *Proc. of the Software Engineering Environments '93, Reading, UK*, 1993. To appear.
7. W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments — The Goal has not yet been attained. In I. Sommerville, editor, *Proc. of the* 4^{th} *European Software Engineering Conference, Garmisch-Partenkirchen, Germany*, Lecture Notes in Computer Science. Springer, 1993. To appear.
8. G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments — Part 1: Tool Specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.
9. F. Gallo, R. Minot, and I. Thomas. The object management system of PCTE as a software engineering database management system. *ACM SIGPLAN NOTICES*, 22(1):12–15, 1987.
10. M. Kampmann. Werkzeuge zur Unterstützung gruppenorientierter Arbeit beim Softwareentwurf. Master's thesis, University of Dortmund, Dept. of Computer Science, January 1993.
11. W. Kim, N. Ballou, H.-T. Chou, J. F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 251–282. Addison-Wesley, 1989.
12. P. Kroha. *Objects and Databases.* McGraw-Hill, 1993. To appear.
13. C. Lewerentz and A. Schürr. GRAS, a management system for graph-like documents. In *Proc. of the* 3^{rd} *Int. Conf. on Data and Knowledge Bases.* Morgan Kaufmann, 1988.
14. M. A. Linton. Implementing Relational Views of Programs. *ACM SIGSOFT Software Engineering Notes*, 9(3):132–140, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.
15. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator – a system for constructing language based editors.* Springer, 1988.

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