Dedicated Object Management System Benchmarks for Software Engineering Applications

Wolfgang Emmerich and Wilhelm Schäfer
Informatik 10
University of Dortmund
P.O. Box 500 500
D-4600 Dortmund 50
emmerich@udo.informatik.uni-dortmund.de
wilhelm@udo.informatik.uni-dortmund.de

Abstract

Non-standard database systems become available now, even as commercial products. They overcome a lot of deficiencies of relational systems w.r.t. their use in engineering applications like computer-aided design or software engineering. Their rather sophisticated functionality especially concerning the manipulation of complex objects makes them highly attractive for engineering applications. If being used as the central database of a rather complex application they could however still become a bottleneck w.r.t. performance. This paper presents a new way how to define a special purpose benchmark which enables to select the fastest database system for a particular software engineering application. It is argued that existing benchmarks are not appropriate to support such a selection, because they neglect important application specific characteristics which significantly influence the database performance.

1 Introduction

Integrated Software Development Environments (SDEs) include a number of tools which support most of the life-cycle phases and inform, analyse, and check document interdependencies and sometimes even propagate changes across document boundaries. The relation between different documents is either based on a transformational approach, e.g. information from a requirements specification is automatically extracted and used as a skeleton for the design document, (cf. ProMod [Hru87]) or in case of a more sophisticated functionality, an SDE enables the incremental intertwined, and syntax-directed development and maintenance of all documents. In the latter case, the environment can easily trace back errors through different documents and propagate necessary changes to correct the errors. Examples for such environments are Gandalf [HN86] and IPSEN [ELN+92]. Many more can be found in [Hen88] and [Tay90].

In any case a large number of objects on very different levels of granularity have to be stored and maintained [Pen87]. Of course, the more fine-grained objects are being stored, the more sophisticated functionality in terms of an incremental intertwined development of documents can be achieved [ELN+92]. The more critical however becomes the performance of an underlying data store, which is in any case the key component of an integrated SDE. For instance, an appropriate fine-grained data model for any type of document is an abstract syntax graph [MR91, ENS87]. Such a fine-grained model, if being used as the basis for the conceptual schema of an SDE's central data store, supports the construction of syntax-directed tools and especially allows the expression of document interdependencies on the fine-grained level of syntactical units like identifiers, functions, operations, interfaces, sections, etc. which in turn enables analysis, error detection and change propagation on that fine-grained level even across document boundaries. Because of those advantages, the support of a fine-grained data model is a major requirement for our performance investigations.

It is then very obvious that the use of a specific database system could very quickly become the performance bottleneck of an SDE. In most cases the required response time must be below one second in order to provide a user friendly system. It has also become clear that the relational database technology does not address the requirements of database systems for SDEs appropriately. Rationales for the statement in the last sentence can be found in [Mai89] and [LS88]. Therefore, a number of development efforts have been started to build dedicated so-called non-standard database systems for software engineering applications or related areas. (CAD-environments or hypertext systems have very similar requirements especially with respect to the granularity of the data model and performance.
It is worthwhile to note here that the same arguments not only hold for fully fledged SDEs, but also for single syntax-directed tools. (That is the reason why the title of this paper mentions software engineering applications in general.) As the amount of data produced by single tools is significantly smaller than in the case of an SDE, the performance issue may not become that critical, but in principal, the required functionality and the requirements for the granularity of the data model are the same.

A number of non-standard database management systems (OMSs)[1], are now available either as academic prototypes like GRAS [LS88], P Graphite [WWFT88] or already as commercial products like PCTE/OMS [GMT87], GemStone [BMO$^+$89], O_2 [BDK91], and many others. They still differ significantly particularly in respect of the provided functionality and the data model which is the basis for defining the documents' internal representation within the OMS.

The simple question which this paper wants to answer, is: how to find the right OMS? In more detail, this paper provides a strategy for selecting the most appropriate OMS for a particular application, i.e. a special set of tools integrated within an SDE. The most dominant selection criterion is, of course, database performance, as it is finally a major acceptance criterion for the SDE built on top of the OMS.

The problem in selecting the most appropriate OMS is that the usual procedure of running a standardised benchmark does not work for OMSs. The above mentioned heterogeneity of the provided data model, and the provided programming interface prevents the definition of a uniform benchmark as a piece of source code like the Dhrystone benchmark for compiler- and operating system-performance [Wei84] or the Wisconsin benchmark for relational database systems [DeW91]. OMSs do not have a standardised programming interface like the POSIX standard for operating systems nor do they provide a standardised query language like SQL.

The benchmark therefore has to be defined on a more abstract level than source code. It will

[1] Most of these newly developed systems fall into a category which is often also called Object Management Systems (OMS is used synonymously for object-oriented database systems. Concerning the notions we refer to [Dit86]). We think that the development of OMSs as the basis for SDEs is the most promising direction to go and we have therefore concentrated our investigations on these systems so far. The approach described in this paper is however independent from any particular system functionality.

basically be a conceptual schema plus a number of operations based on the entities defined in the schema (e.g. insert and delete operations). In addition, the particular values of the operations' parameters have to be defined and furthermore one or more initial database states. It then has to be implemented on top of the OMSs under investigation.

This paper is further structured as follows. The next section briefly introduces two existing OMS benchmarks which have not been defined for software engineering applications. Their appropriateness for software engineering applications is discussed in section 3. Section 4 presents our approach of application specific benchmarks and illustrates the approach by describing a dedicated benchmark to support the selection of an OMS as the basis for the development of a particular SDE. Section 5 describes major aspects of the implementation of benchmarks. Section 6 concludes the paper by sketching some general lessons learned about using OMSs from the implementations of a number of benchmarks on top of different OMSs.

2 Related Work

Based on the above sketched arguments that benchmarks cannot be defined always in terms of source code, two so-called "abstract OMS benchmarks" have been defined so far. The first one is the so called "simple Benchmark" described in [DEL92]. The second one is the "Hypermodel Benchmark" defined in [ABM+90].

2.1 The simple Benchmark

The simple benchmark was defined in order to measure the performance of elementary OMS operations. The conceptual schema for this benchmark (as well as for the others in this paper) is shown as an Entity-Relationship diagram (E/R-diagram) in figure 1.

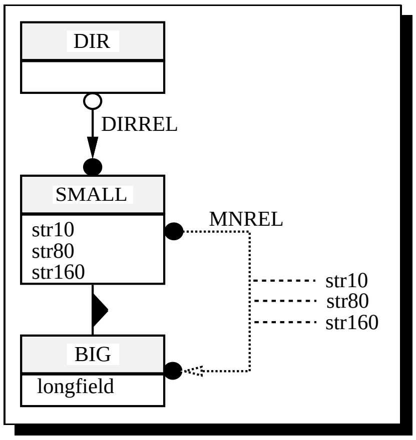


Figure 1: E/R-Diagram for the simple benchmark

In these diagrams, a rectangle models an entity type. A solid arrow between entities represents an aggregation type relationship. Its semantics is that no entity can exist without being related in an aggregation to an already existing entity, i.e. the aggregation relationship models the part-of/belongs-to relationship. Dotted arrows model reference type relationships. At the end of arrows, black circles represent a many-end of a relationship, and white circles represent an one-end. A circle placed on top of a line declares the relationship to be ordered. Attributes of entities are described within a type definition whereas attributes of relationships are shown besides the resp. arrow. A triangle on a line between types defines an inheritance relation meaning that a sub-entity inherits all attributes its super-entity holds and all relationships it participates in. Multiple inheritance is not allowed.

The above schema defines entities of types *DIR*, *SMALL*, and *BIG* and relationships of types *DIRREL* and *MNREL*. Entities of type *DIR* are used to connect entities of types *SMALL* and *BIG* via an 1:n relationship of type *DIRREL* to the database. Entities of type *SMALL* and *BIG* as well as the instances of the m:n relationship *MNREL* have attributes which allow the storage of strings of the lengths 10, 80, and 160 bytes. Entities of type *BIG* have an additional attribute *longfield* which is dedicated to the storage of byte streams of the lengths 10 and 128 kbytes.

The operations defined by the simple benchmark include creating and deleting small and big entities, as well as creating and deleting relationships between them. Furthermore operations on attributes of entities and relationships such as storing and retrieving strings of lengths 10, 80, and 160 bytes or longfields of the lengths 10 and 128 kbytes are defined.

It is part of the benchmark that the operations access and modify a non-empty data. This avoids that objects accessed by the operations reside only in OMS caches (i.e. in main memory). A realistic size of an initial database defined before performance measurements start, guarantees that operations have to access secondary storage (as it usually happens in real applications). The simple benchmark defines the size of the initial database to contain 3,000 entities of type *SMALL* and 400 entities of type *BIG*.

2.2 The Hypermodel Benchmark

The Hypermodel benchmark differs from the simple benchmark in using more complex data structures and operations. The benchmark is a development dedicated to hypertext applications.

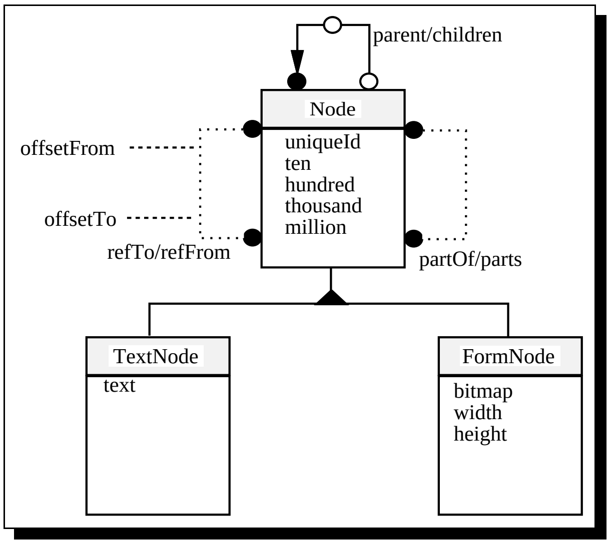


Figure 2: Conceptual schema of the Hypermodel benchmark

The conceptual schema of the Hypermodel benchmark is shown in figure 2. It defines three different kinds of entity types, namely *Node*, *TextNode*, and *FormNode*. *TextNode* and *FormNode* are subtypes of *Node*. Entities of type *Node* represent sections of a hypertext which are further structured. Entities of type *TextNode* represent an unstructured text and entities of type *FormNode* represent a bitmap. Three types of relationships are defined, namely the *parent/-children* relationship, the *partOf/parts* relationship, and the *refTo/refFrom* relationship. The *parent/children* relationship is of cardinality 1:n, it is ordered, and it defines the aggregation structure between nodes. The m:n *partOf/parts* relationship models the section/subsection structure of a hypertext and the m:n relationship *refTo/refFrom* models arbitrary hypertext links. Each entity of type *Node* has five attributes called *uniqueId, ten, hundred, thousand,* and *million.* Additionally, an entity of type *TextNode* contains a text attribute *text* and a *FormNode* has three attributes *width* and *height* to store the dimensions of a picture and a longfield attribute *bitmap* to store the picture itself. Furthermore, the *refTo/refFrom* relationship contains two attributes *offsetFrom* and *offsetTo* which does not only allow a description of the source- and target-node of a hypertextlink but also enables to define its exact positions within the related *text* attributes.

The initial database contains a completely balanced tree of varying depth built of nodes and *father/children* relationships. Each inner node is of type *Node* and has exactly five children. Each leaf node is either of type *FormNode* or *TextNode.* The *partOf/parts* relationship is created for each node by selecting one inner node of level k and relating it to five random nodes from level k+1. The *refTo/refFrom* relationship is created for each node to another random node. Nodes are numbered and the number of a node is stored in the *uniqueIds* attribute. *Ten, hundred, thousand,* and *million* are initialised by random numbers selected from the corresponding interval. Each attribute of entities of type *TextNode* is initialised with a text containing a number of up to 100 words each having up to 10 characters. A formnode consists of a random square bitmap with an edge length of up to 400 pixels.

The operations of the Hypermodel benchmark include mainly retrieval operations such as lookups for attributes with particular names or values in particular ranges, lookups for node sets connected by the above mentioned relationship in normal or reverse order, and finally groups for operations performing a sequential scan and a transitive closure traversal following different relationships. The only update operations substitute words in the *text* attribute of a text-node and inverts a subrectangle within the bitmap attribute of a randomly selected formnode. The detailed description of the operations is of no concern for the scope of this paper. The interested reader is referred to [ABM$^+$90].

3 Why Application specific Benchmarks

This section argues why the formerly presented benchmarks are not appropriate in software engineering applications and indicates how this deficiency could be remedied.

The main problem of the simple and Hypermodel benchmark is that their conceptual schemas are too simple to meet an SDE application. Consider, for example, an SDE that includes syntax-directed tools supporting the development of structured analysis (SA) diagrams, modular designs and programs written in a usual programming language. Concerning the SA tool, the structure of data-flow diagrams, a data-dictionary, and the mini-specifications should be reflected in the conceptual schema of the benchmark. Concerning the design tool, entity types reflecting the structure of modules have to be defined. Concerning the programming environment, the schema has to contain a large number of entity types reflecting the very fine-grained

structure of syntactic increments of the underlying programming language. Those entities mentioned would differ significantly w.r.t. the number, type, and size of their attributes. Some entities, for instance, would have to carry just small attributes for defining graphical coordinates, others like mini-specifications or comments would carry only long-field attributes to store text, and finally other entities' attributes would be a combination of the two previous ones.
Those examples should just indicate the **heterogeneity** of possible **entity types** in an SDE as opposed to the homogeneous definition of entity types in the two mentioned benchmarks. The simple benchmark defines just two different types of entities and the Hypermodel benchmark three different types.
Unfortunately, the number and size of attributes of an entity type significantly influence the time necessary to retrieve and create objects of that type. Hence, the sketched heterogeneity of types would have to be reflected in a benchmark addressing performance requirements of an SDE.

Besides the large number of possible entity types, different relationships have to be reflected in the conceptual schema as well. One to many composition relationships define how complex objects are aggregated from simpler ones. In SDEs it happens frequently that objects of different types are aggregated to one complex object. We call that a **heterogeneous aggregation**. As examples for this kind of aggregations consider the import section of a module which may be composed of imported types, procedures, and functions or a data-flow diagram that consists of nodes, terminators, stores, and data-flows. Some of these heterogeneous aggregations are ordered (e.g. imports), whereas others are not (e.g. data-flow diagrams). Moreover these composition relationships sometimes form **nested aggregations**. As examples consider nodes of a data-flow diagram that are refined and hence consist of other data-flow diagrams or procedure declarations that may contain further procedure declarations.
The simple benchmark schema contains neither heterogeneous nor nested aggregations. In the Hypermodel benchmark only the parent/children relationship could be nested but heterogeneous aggregations also do not exist.
However, access to an object in a homogeneous aggregation turns out to be much faster than to an object in a heterogeneous aggregation. Moreover, nested aggregations, of course, require significantly more time for navigation than allocation of a particular object in a flat structure.

In addition to the mentioned deficiencies of the aggregation relationships, both benchmarks do not adequately address the software engineering application requirements w.r.t. additional so-called reference relationships. Both define reference relationships which however do not map situations frequently occurring in SDEs.
In case of the simple benchmark, the reference relationship *MNREL* is instantiated by the initial database and by benchmark operations as if it were a 1:1 relationship, i.e. the benchmark connects one entity of type *BIG* with one of type *SMALL*. In case of the Hypermodel benchmark, the *partOf/parts* relationship links a node with exactly five other nodes and the *refTo/refFrom* relationship relates a node with exactly one other node. In an SDE, however, a reference relationship is usually of cardinality 1:n where n tends to become rather large, namely up to a few hundred. As an example, consider a basic type identifier in a large software system. It will be used in a large number of modules by a large number of operations as parameter or result type. All entities representing the use of this identifier must be linked by a 1:n relationship to the entity representing the declaration.

Operations of the benchmark should be defined based on the type definitions in the conceptual schema. This includes insert and delete operations for each entity and each defined attribute.

This requirement is fulfilled by the simple benchmark whereas the set of operations of the Hypermodel benchmark only includes two update operations which just change attribute values.

Furthermore, simple benchmark operations just as the conceptual schema itself suffer from their simplicity. The functions offered by a tool of an SDE are usually composed of a number of database operations which could correspond to operations as defined in the simple benchmark. Unfortunately just summing up the execution time of these simple operations gives wrong results. As a matter of fact, complex tool operations can sometimes be implemented much more efficiently by exploiting a particular feature of the OMS under investigation than by just taking a particular order of predefined simple benchmark operations.
Some OMSs, for example, provide a special type called dictionary and a corresponding member function (which of course can be assumed to be implemented in a very efficient way by the OMS developers). If this type and especially the member function had been implemented by a number of simple benchmark operations, this would result in a much higher execution time than if using the member function.

Finally, none of the benchmarks has considered the consequences of concurrent execution of database operations yet. That includes the definition of one or more transaction concepts (like optimistic or pessimistic models) and its corresponding realisation by the operations defined by a benchmark.

The initial database state in a benchmark definition should reflect realistic situations of the application. Neither the simple nor the Hypermodel benchmark meet this requirement in the case of an SDE. The static and simple definition of the simple benchmark does not at all reflect situations which appear in SDEs. The same is true for the Hypermodel benchmark. Even though it defines more than one initial state, all of them define a completely balanced tree which is a very unusual situation in SDEs.

Besides clarifying the deficiencies of the existing benchmarks the above examples are supposed to indicate that even different SDEs could have very different performance requirements, i.e. it is impossible to define a general benchmark for evaluating OMSs for SDEs. In more detail, the tools and types of documents in an SDE determine the entity types and their relationships which vary significantly depending on the particular tools included in the environment. As mentioned, storage requirements for a structured analysis diagram are very different from the requirements for the storage of programs. In addition, the consistency constraints and dependencies between different documents are very important for the definition of a benchmark. In an SDE following a transformational approach a much less fine-grained data model is needed than in an SDE enabling intertwined document development. This in turn results in a much lower number of relationships for the transformational case. Furthermore, the initial database state depends on a particular application, i.e. tools, document types, and even the scale of projects being performed with the SDE.

Our approach therefore is not to extend the benchmarks mentioned to additionally meet (some) requirements from SDEs nor to develop another general benchmark for the area of SDEs. We rather propose to develop application specific benchmarks for a special SDE, i.e. for a particular set of document types, and a corresponding particular set of tools. Of course such an approach should still enable a systematic development of an application specific benchmark based on reusing as much information as possible from previous experiments. We do not want to build a special benchmark for any new set of tools and documents from scratch again. We propose an organised, carefully designed process to define an appropriate benchmark which includes the steps (1) to define an appropriate benchmark based on the requirements of a particular application, (2) to implement it on top of the OMS under investigation, and (3) to reuse as much information and code as possible from previously developed benchmarks. This approach is described in the next two sections.

4 Definition of an Application Specific Benchmark

The overall objective of the definition of the benchmark is to derive a conceptual data (or object resp.) base schema and the corresponding retrieval and update operations.
This section describes this derivation process by identifying the different steps and their respective input and output. In addition each step is illustrated based on a particular example, namely the selection of a suitable OMS for the Opus SDE. This example is called the Opus benchmark.

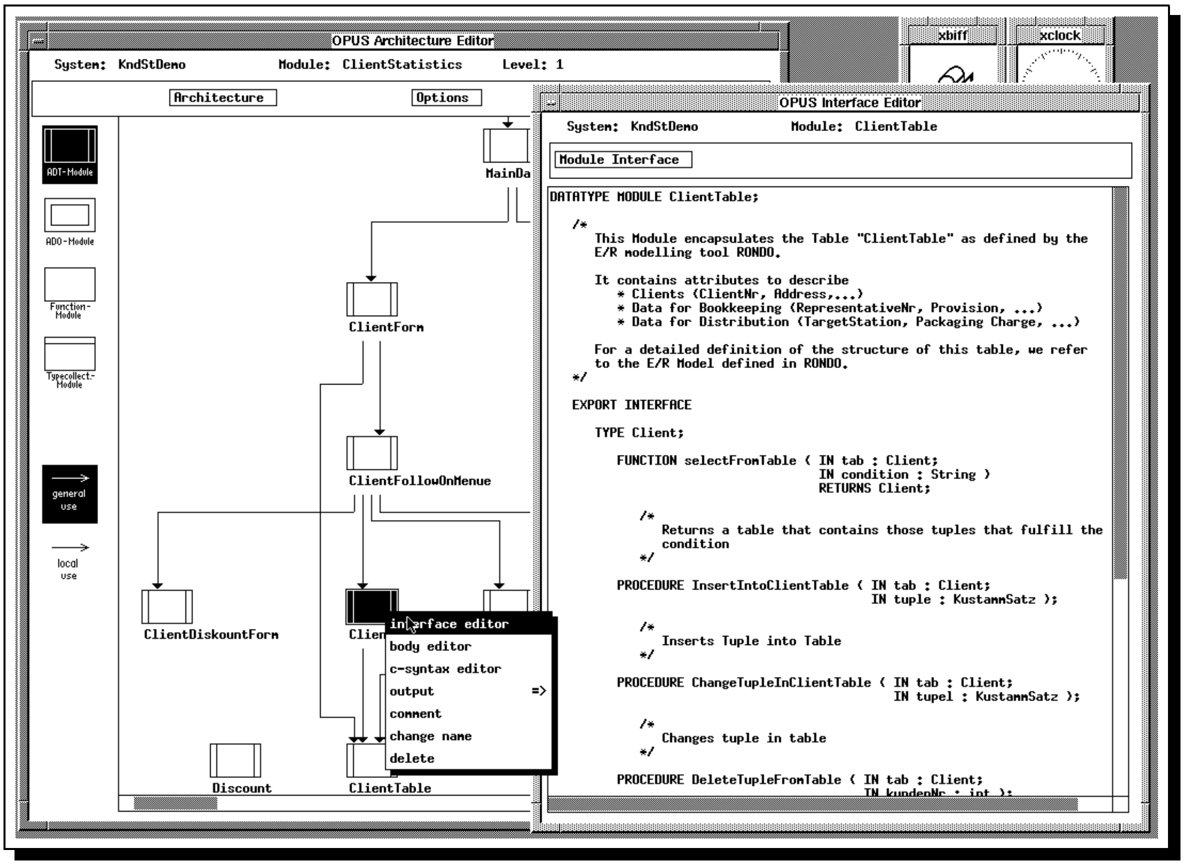


Figure 3: User Interface of the OPUS SDE

The Opus SDE consists of two highly integrated syntax-directed design and specification tools. Those tools support the development of a modular-like architecture in terms of modules, module interfaces, and different types of use-relations between modules. Fig. 3 sketches the user interface of this SDE. A more detailed description of a previous version of this environment can be found in [Lew88]. However, it is of no concern for the scope of this paper.

4.1 The conceptual Schema

The derivation of the database schema starts from taking the syntactic definition of each document whose development is supported by a particular SDE. (Remember that our goal is to identify the most suitable OMS for a particular SDE.) The syntactic definitions are usually given or at least can be transformed into a tree grammar representation, defining the abstract syntax of each document (c.f. [BCD+88]). This is what we need as the basis for our benchmark definition.

```
Fixed arity operators:

arch        -> ARCH_ID MOD_LIST
module      -> MOD_ID  COMMENT  IMP_LIST

Atomic operators:

arch_id     -> IDENT
mod_id      -> IDENT
comment     -> STRING

List operators:

module_list -> MODULE ...
import_list -> IMPORT ...

Phyla:

ARCH_ID     :: arch_id
MOD_LIST    :: module_list
MODULE      :: module
MOD_ID      :: mod_id
IMP_LIST    :: import_list
IMPORT      :: mod_id
COMMENT     :: comment
```

```
Fixed arity operators:

module      -> MOD_ID   COMMENT  EXPORT_PART  IMPORT_PART
export      -> TYP_ID   OP_LIST
func        -> OP_ID    PAR_LIST TYP_ID  COMMENT
proc        -> OP_ID    PAR_LIST COMMENT
cbv_par     -> PAR_ID   TYP_ID
cbr_par     -> PAR_ID   TYP_ID
import_part -> IMP_LIST
import      -> MOD_ID   IMP_OBJ_LIST

Atomic operators:

mod_id      -> IDENT
typ_id      -> IDENT
op_id       -> IDENT
par_id      -> IDENT
comment     -> STRING

List operators:

op_list     -> OP     ...
import_list -> IMPORT ...
par_list    -> PAR    ...
imp_obj_list-> IMP_ID ...

Phyla:

MOD_ID      :: mod_id
COMMENT     :: comment
EXPORT_PART :: export
IMPORT_PART :: import_part
OP_ID       :: op_id
TYP_ID      :: typ_id
PAR_ID      :: par_id
OP          :: func proc
PAR         :: cbv_par cbr_par
IMP_LIST    :: import_list
IMPORT      :: import
IMP_OBJ_LIST:: imp_obj_list
IMP_ID      :: typ_id op_id
```

Figure 4: Abstract Syntax for Opus Benchmark

Taking our Opus example, the abstract syntax of two document types is defined. An excerpt of its tree grammar definition is shown in figure 4. The first document type allows to define modules and their import-relationship. The second type is dedicated to the detailed definition of the export- and import-interfaces of modules. In particular, it allows the declaration of types, procedure heads, and function heads exported by a module and refines the import-relationships from other modules by allowing the definition of imported objects for each relationship (c.f. also figure 3.

Based on the transformation rules given in table 1 it is a straightforward exercise to derive a conceptual schema given as an Entity-Relationship (E/R) diagram from a tree grammar. E/R diagrams are one of the most commonly used languages to define conceptual schemas. As, especially, other benchmarks are also given by this notation (cf. section 2), we use this notation as a common basis thus enabling an easy comparison of our benchmark definition with the previous ones and maybe upcoming new ones. Applying the transformation rules of table 1 to the tree grammar excerpt in fig. 4 results in the E/R diagram in fig. 5.

The next step is to extend the schema by additional relationships. Those relationships are the basis for holding consistency constraints between entities when modifying the database. Those constraints include, for example, uniqueness of identifiers, dependencies between declaration of identifiers and their usage, etc. i.e. the constraints concern the static semantics of a language definition as well as interdocument consistency. Not including such relationships into a schema would result in significant runtime increase of all update operations based on the schema definition.

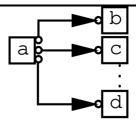
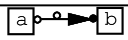
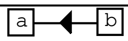
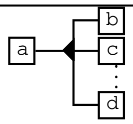
For tree grammar component of type	Substitute $\boxed{\texttt{a}}$ with
Fixed arity operator `a -> B C .. D`	
List operator `a -> B ...`	
Atomic operator `a::=B`	
Phyla `a::B C ... D`	

Table 1: Transformation of tree grammar into E/R diagrams

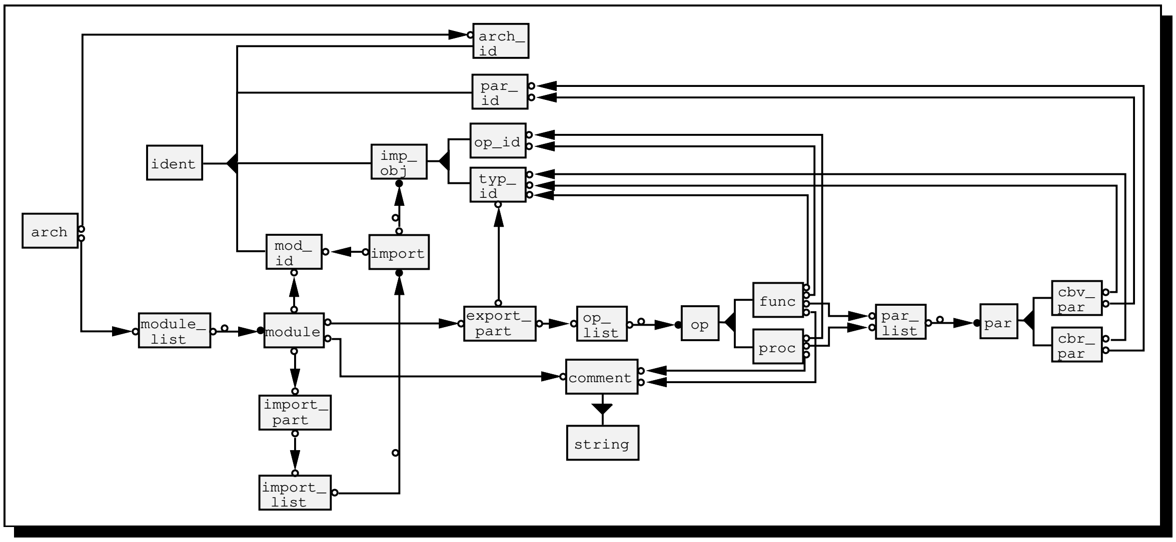


Figure 5: E/R diagram deduced from Grammar

The additional relationship types to be introduced are all of cardinality one to many. They serve three different purposes, namely (1) they support associative queries which, for example enable to quickly determine whether a particular identifier has been introduced, (2) they support object sharing, e.g. the name of an identifier is only stored once and accessed from different places where it is being used, (This avoids redundant information in the database and thus it avoids complex update procedures.) and (3) they define change propagation paths between different objects which are especially helpful when interdocument consistencies have to be updated.

As an example, the constraints defined by the Opus benchmark are that

1. the interfaces of modules which occur in an architecture diagram, are specified in the specification document and vice versa,

2. each import interface relationship in the architecture is specified in detail in the specification language and vice versa,
3. names of modules, types, functions and procedures are unique within an architecture diagram and all related specification documents,
4. modules which participate in an import-relationships, have to exist,
5. objects which are imported by an import-relationship, are exported elsewhere,
6. cyclic import relationships are forbidden,
7. types used in parameters of operations and result types of functions are declared, i.e. they are either exported by the module in which they are used, or imported from elsewhere.

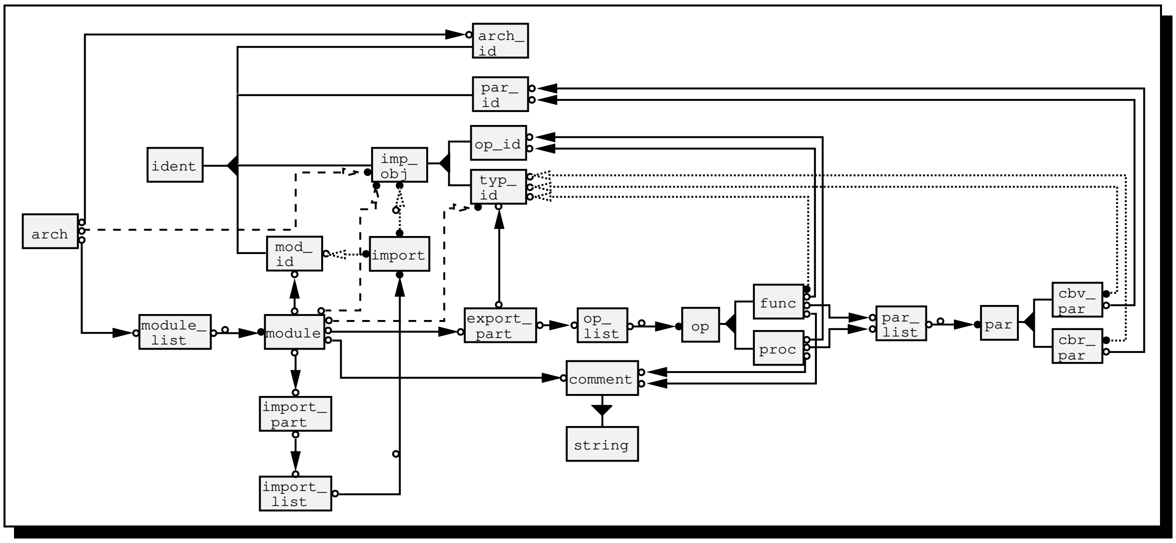


Figure 6: E/R Model enhanced with context sensitive Relationships

The result of adding relationships according to those constraints to the E/R diagram of figure 5 is given in figure 6. The first additional relationship type, which we call *dictionary*, is drawn using dashed lines, whereas the second relationship type, which we call *reference* is drawn using dotted lines.
One dictionary relationship defines all identifiers which are declared in an architecture. Another dictionary relationship defines all the identifiers which are exported by a module, i.e. that may be used as imported objects. The last dictionary relationship defines all type identifiers that may be used in a module.
To avoid change propagations reference relationships which substitute aggregation relationships, allow object sharing. In particular, the types used in parameters of operations, or result types of functions are no longer viewed as copies of types defined in export interfaces, but as references to them. Furthermore, the copies of identifiers of imported modules and objects in import lists are transformed into references to the resp. identifiers. This not only allows omitting of time consuming change propagations, but also enables quick checks whether an exported type or operation is actually used.

The next and a major step is a simplification of the schema defined so far. This simplification results in a schema which has to be understood as a schema for a "real" benchmark, because the schema derived in the previous steps was basically the documents' syntax representation including static semantics information.

The simplification is defined by a number of rules. The application of those rules removes all entities and relationships which do not influence the performance of benchmark operations because the entities remaining in the schema represent the worst case situation. The rules for schema simplification are that

1. relationships which start from or end in all sub-entities of an inheritance relationship, are replaced by one relationship which starts from or ends in the super-entity,
2. entities which do not participate in any relationship except if they are the target of an aggregation type relationship, are transformed into attributes of the entities, where the aggregation relationship starts,
3. sub-entities of an inheritance relation, which participate in the same relationships and carry the same attributes as another entity of that inheritance relation are removed. The remaining entity is then viewed as a placeholder for the removed entities. As a consequence, execution times of benchmark operations that access this entity must rather be interpreted as upper bounds than as exact values of operations that would have accessed objects of the removed entity type,
4. an inheritance relationship with only one sub-entity is removed together with its sub-entity. The super-entity subsumes all relationships, the sub-entity participated in, as well as all the subentity's attributes,
5. an entity that neither participates in a context-sensitive or inheritance relationship nor carries attributes and which is the source of only one aggregation relationship, is removed. The aggregation relationship that started from the entity now starts from each entity that had an aggregation relationship to the removed entity.

The ordering in which the simplification rules may be applied to the E/R diagram is as follows: Rules 1–4 may be applied repeatedly in mutual exclusion, rule 5 is applied repeatedly only after the application of rules 1–4 is finished.

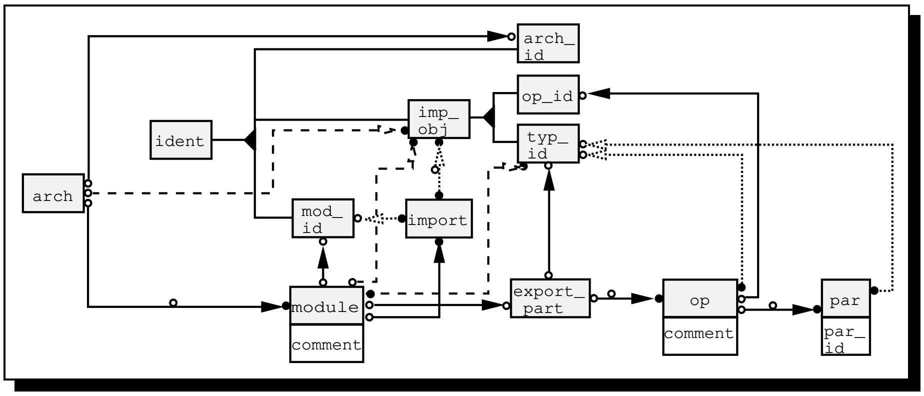


Figure 7: E/R Model of simplified Object Base Schema

Using these simplifications we are able to simplify the E/R diagram shown in figure 6. The result of this process is depicted in figure 7. We applied rule 1, 3, and 4 to `cbv_par` and `cbr_par` with the effect of removing these entities and transferring its objectives to `par`. Then we were able to apply rule 2 to `par_id` transforming it into an attribute of entity `par`. After that, we applied rule 3 to `func` and `proc` with the effect of removing the entity `proc`. That enabled us to apply rule 4 to `func`, with the effect of replacing entity type `func` by its super-entity `op`. As `comment` is only the target of aggregation relationships, we could apply rule 2, thus transforming `comment` into attributes of `module` and `op`. The same rule was applied to entity `value` connected to `ident` transforming this entity into an attribute of `ident`. Finally the entities `module_list`, `import_part`, `import_list`, `op_list`, `par_list` have been removed according to rule 5.

4.2 The initial Database

The next step in defining a benchmark is the definition of an initial database (cf. section 2). In order to determine a realistic structure and a realistic number of objects, we perform an analysis of existing documents. We assume that those documents exist. Usually SDE developers have gained preliminary experience with the document types while they were producing documents during case studies performed with text editors or syntax-directed editors which could be easily generated using a generator like e.g. Centaur [BCD$^+$88].
Based on the benchmark schema, we must obtain average values for the number of entities for each 1:n relationship. Moreover, we have to obtain average sizes for the attributes defined in the schema.

It is usually possible to acquire the existing documents into a text-file. Then the analysis can be performed by a parser generated by e.g. `lex` and `yacc`.
In contrast to the Hypermodel benchmark, this approach leads to initial databases that have a structure similar to that of real documents.

Metric component	Value
Number of exported types	1
Number of exported operations	17
Number of imported modules	4
Number of identifiers	132
Number of comments	18
Number of imported objects/import relationship	6
Number of parameters per operation	3
Length of identifiers [bytes]	12
Length of comments [bytes]	256

Table 2: Metric for a module in the initial object base

The structure of the initial database for the Opus benchmark is based on the analysis results of some 5,000 lines of specification produced when specifying the Opus SDE itself. Table 2 defines the number of components of a module contained in the initial database according to the schema defined in figure 7. To vary the size of the initial object base we increase the number of modules by increasing the number of levels in the architecture as follows: The import-relation between modules leads to an acyclic graph of modules. We divide the modules into n levels ($n \geq 3$). Each level L_i contains 2^i modules ($i \in \{0, \ldots, n-1\}$). Except the top-most level where a module imports from both modules at level 1, a module in level L_j imports from four random modules of level L_{j+1} ($j \in \{1, \ldots, n-2\}$).

4.3 The Operations

The final step in defining a benchmark is the definition of the operations and especially the values of their parameters. It follows the guidelines given for the operation definition in section 3. The main point in defining parameter values is to keep the structure of the initial database. Thus, the definition of those values is also based on the above mentioned analysis results.

The Opus benchmark operations are clustered into four groups. Operations of the first group create entities of the benchmark schema like modules, types, operations, parameters, and comments. The second group is dedicated to measure the impact of changes to these entities. The third group simulates tool operations that perform analysis such as *show all modules that import from a particular module*. Finally, the last group deletes all previously created entities. Meeting the consistency constraints defined above is always assured by the operations of the Opus benchmark.
Based on the value depicted in table 2 the parameter values for expanding an operation list has, for example, been defined to create 17 new operations.

5 Implementation of an abstract Benchmark

The implementation of a benchmark has to be done in such a way that the benchmark performs as fast as possible, i.e. the implementation must exploit the functionality of an OMS as much as possible in order to decrease runtime. This leads to a dilemma, because we have to know about the performance of an OMS, before we actually finished the implementation of a benchmark. To solve this dilemma, we assess the performance of the elementary functionality of an OMS using the simple benchmark. The results of this activity enable to decide later on which elementary functions should be used when implementing the more complex functionality of an application specific benchmark. For instance, two elementary functions to implement associative access to sets of objects have to be distinguished with respect to performance. In this example it turns out that the function implemented by external hashing has a better performance than the one implemented by B-trees.

Finally, a benchmark implementation as described above is in itself a rather complex software development activity. If it is done following good software engineering practice, a modular design of the benchmark implementation should precede the implementation phase. Such a design later on enables to reuse major parts of the code for investigating other OMSs.

The overall design of a benchmark implementation is given in figure 8. Each box depicts a subsystem composed of modules and the arrows indicate the usage-relationship between them. The top-level module coordinates the execution sequence of the benchmark operations and therefore calls the operations in the *Benchmark operations* subsystem. That subsystem coordinates the execution of the defined benchmark operations and recording the time used for their execution. It therefore uses the *Time measurement* subsystem to measure execution-times and log them in a file. It uses the *internal data* subsystem which maintains internal data structures for object addressing purposes by references to objects stored in the OMS. Finally, it uses the *Increment operations* subsystem which consists of the implementation of any single benchmark operation based on the OMS programming interface.
Following this suggestion during the implementations of the Opus benchmark on top of a number of OMSs, we were able to reuse about 40% of the code.

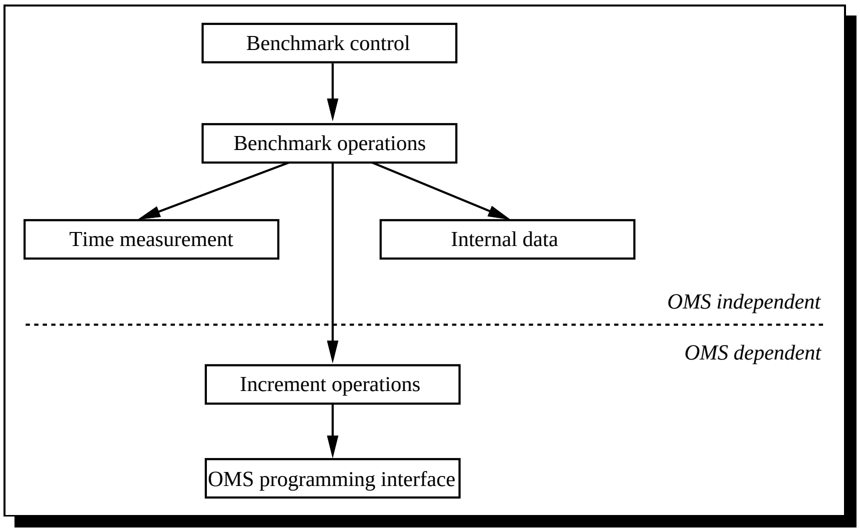


Figure 8: Architecture for OMS Benchmark Implementations

6 Concluding Remarks

The work described in this paper arose from the implementation of a number of stand-alone CASE tools and complete SDEs. These developments were research prototypes. Some of them have been turned into commercially sold tools based on a collaboration contract with STZ GmbH, a medium-sized Dortmund-based software house. When the development of those tools started, OMSs were not available in the market. Thus the development started on a self-made OMS called GRAS [LS88] which is a dedicated storage system for storing and retrieving arbitrary large graphs, i.e. GRAS especially supports a fine-grained data model.

For several reasons, in particular portability across operating systems, safety and multi-user support, most of the tools are being ported or have to be ported on a new platform, namely one of the now available modern OMSs. The benchmark described here helped us and will help us to select between a number of different offers. Systems investigated so far, include the first platform GRAS, but also GemStone [BMO+89], PCTE/OMS [GMT87], VBASE [AH87] CadLab/OMS [GKK+87], Damokles [DGL86] and Object-Base [WS87].

The implementation of benchmarks on top of several OMSs not only supported our above mentioned selection process, but it also gave us a number of useful insights on how to exploit an OMS in general as an SDE platform. The experience in building SDE database platforms on top of OMSs will be briefly summarised here. A more detailed description will be subject of a forthcoming paper.

In order to support particularly the construction of syntax-directed tools, an OMS must provide the access to any object via a unique identifier or surrogate, usually called object identifier (OID). Other existing possibilities to access objects, namely via pathnames which define a path from a common root via a number of objects to a particular one or relational queries do not perform fast enough to support especially syntax-directed editing

Other tools than editors like, for example, browsers or analysers require the possibility of an associative database query. Typical questions in that respect are: "Is an identifier's name unique" or "Show all places where a particular identifier is used". A number of OMSs offer the possibilities of either implementing such queries based on B-trees or an external hashing. (For example, GemStone distinguishes between Sets and HashDictionaries). Our experience strongly suggests that the implementation of such an associative query must be based on external hashing in order to achieve acceptable performance.

An OMS must further provide an object caching mechanism such that the relevant parts of the usually large abstract syntax trees (or graphs resp.) are paged. OMSs not providing such a mechanism like e.g the only commercially available PCTE/OMS implementation do not offer adequate response time especially for implementing editing operations. The caching must be supported by an adjustable clustering mechanism, i.e. the SDE developer must be able to define which objects reside on a page. This allows to locate objects on one page which are frequently accessed together (which is an application specific issue) and thus the page transfer rate between cache and secondary storage is significantly decreased.

Finally, different OMSs offer different transaction management strategies in order to support safety and multi-user access. The general strategy to have safe applications (in case of sudden system crashes) is to consider any single tool operation like an editing operation as a transaction. Our experience shows that in case an optimistic transaction strategy is used, no significant performance bottleneck arises. However, in order to support multi-user applications, pessimistic locking protocols are needed on a more coarse-grained level e.g. on the level of complete documents.

This aspect has not yet been included in our current benchmarks. Future work will concentrate on introducing performance measurement of multi-user accesses to coarse-grained objects into our benchmarks.

Further work not only focussing on benchmark development but on building a special SDE platform will be carried out as part of a new ESPRIT-III project called GOODSTEP (General Object-Oriented Database for Software Engineering Processes). Its goal is to build an SDE platform based on a particular OMS, namely the O_2 system.

Acknowledgements

We are grateful to our colleges Dr. D. Schmedding, P. Lago, G. Junkermann, Dr. S. Dewal and Dr. R. Fehling for the intensive discussions about OMS benchmarks and for their comments on earlier versions of this paper. Moreover, we appreciated the discussion with A. J. Berre about OMS benchmarks in general. Finally, we are indebted to F. Buddrus and M. Kampmann for their patient support in implementing the benchmarks.

References

[ABM+90] T. L. Anderson, A. J. Berre, M. Mallison, H. H. Porter, and B. Schneider. The hypermodel benchmark. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proceedings of the International Conference on Extending Database Technology – LNCS 416*, pages 317–331. Springer, Mar. 1990.

[AH87] T. Andrews and C. Harris. Combining Lnaguage and Database Advances in an Object-Oriented Development Environment. In *Proc. of Object-oriented programming systems languages and applications, Orlando, Florida*, pages 430–440, 1987.

[BCD+88] 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.

[BDK91] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufmann, 1991.

[BMO+89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone data management system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 283–308. Addison-Wesley, 1989.

[DEL92] 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.* IEEE Computer Society Press, 1992. To appear.

[DeW91] D. DeWitt. The Wisconsin Benchmark: Past, Present, & Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction processing Systems*, chapter 3, pages 119–166. Morgan Kaufman, 1991.

[DGL86] 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, LNCS 244*, pages 353–371. Springer, 1986.

[Dit86] K. R. Dittrich. Object-oriented database systems: the notion and the issues. In K. Dittrich and U. Dayal, editors, *Proc. of the 1986 Int. Workshop on Object-Oriented Database Systems*. IEEE Computer Society Press, 1986.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building Integrated Software Development Environments. *ACM Transactions on Software Engineering and Methods*, 1, 1992. To appear, also Technical Report 60, University of Dortmund, Dept. of Computer Science, Chair for Software Technology.

[ENS87] G. Engels, M. Nagl, and W. Schäfer. On the Structure of Structure oriented Editors for Different Applications. *ACM SIGPLAN Notices*, 22(1):190–198, 1987. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, Cal.

[GKK+87] K. Gottheil, G. Kachel, T. Kathöfer, H. J. Kaufmann, B. Kleinjohann, E. Kupitz, J. Miller, B. Nelke, F. J. Rammig, B. Steinmüller, and C. White. The Cadlab workstation CWS – an open, generic system for tool integration. In F. J. Rammig, editor, *Proceedings of the IFIP WG 10.2 Workshop on Tool Integration and Design Environments*, Paderborn, FRG, Nov. 1987. Elsevier Science Publishers B.V. (North-Holland).

[GMT87] 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.

[Hen88] P. Henderson, editor. *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* ACM Press, November 1988. In: ACM Software Engineering Notes 13(5) and ACM Sigplan Notices 25(2).

[HN86] A. N. Habermann and D. Notkin. *Gandalf: Software Development Environments. IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[Hru87] P. Hruschka. ProMod – in the age 5. In *Proc. of the* 1^{st} *European Software Engineering Conference*, Strasbourg, Sept. 1987.

[Lew88] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. *ACM SIGSOFT Software Engineering Notes*, 13(5):173–182, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

[LS88] 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.

[Mai89] D. Maier. Making database systems fast enough for CAD applications. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 573–582. Addison-Wesley, 1989.

[MR91] S. Meyers and S. P. Reiss. A System for Multiparadigm Development of Software Systems. In *Proc. of the* 6^th *Int. Software Specification and Design Workshop, Como, Italy*, pages 202–209. IEEE Computer Society Press, 1991.

[Pen87] M. H. Penedo. Prototyping a Project Master Database for Software Engineering Environments. *ACM SIGPLAN Notices*, 22(1):1–11, 1987. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, Cal.

[Tay90] R. N. Taylor, editor. *Proc. of the* 4^{th} *ACM SIGSOFT Symposium on Software Development Environments, Irvine, California.* ACM Press, 1990. In: ACM Software Engineering Notes 15(6).

[Wei84] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, Oct. 1984.

[WS87] H. Weber and S. Seehusen. An object-oriented database system: Object-Base. In R. Mitchell, editor, *Industrial Software Technology*, pages 227–246. Peter Peregrinus, 1987.

[WWFT88] A. L. Wolf, J. C. Wileden, C. D. Fisher, and P. L. Tarr. P Graphite: An Experiment in Persistent Typed Object Management. *ACM SIGSOFT Software Engineering Notes*, 13(5):130–142, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.