The Merlin OMS Benchmark
—
Definition, Implementations and Results

Wolfgang Emmerich, Martin Kampmann
Informatik 10
University of Dortmund
P.O. Box 500 500
D-4600 Dortmund 50
emmerich@udo.informatik.uni-dortmund.de

July 22, 1993

Abstract

The bottleneck of highly integrated software development environments (SDEs) could still be the use of an underlying central data store. An environment is called highly integrated, if the included tools support most of the life-cycle phases and especially inform, analyse and check document interdependencies and even propagate changes on basically any level of granularity. In that case a central data store for documents and document interdependencies is the necessary key information base. A number of so-called object management systems (OMSs) have recently been developed to address the needs of SDEs. In order to support an environment builder to select the most appropriate one, this paper presents a new benchmark for such databases. We precisely define that benchmark in an abstract way in terms of its conceptual schema, possible initial database states and the benchmark operations. The benchmark was implemented twice to allow for a comparison of the OMSs GRAS and GemStone. We describe these two implementations as well as the results we gained when we performed the benchmark.

Contents

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, (c.f. 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 Gandalf [HN86] and IPSEN [Nag85].

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 central store and a key component of an integrated SDE.

It then is very obvious that the use of a specific data base system could very quickly become the performance bottleneck of an SDE especially because in most cases the required response time must be below one second. It has also become clear that the relational data base technology does not address the requirements of data base systems for SDEs appropriately [Mai89, LS88]. Therefore, a number of development efforts have been started to build dedicated so-called non-standard data base systems for software engineering applications or related areas.

A number of non-standard database management systems (OMSs) like object-oriented databases [ABD+90] (e.g. GemStone [BMO+89] or O_2 [BDK92]), databases with a graph-like data model (e.g. GRAS [BL85] or P Graphite [WWFT88]), or databases that have been evolved by extending the UNIX file system (e.g. PCTE/OMS [GMT87]) are now available. They still differ significantly, particularly with respect to the provided functionality and the data model which is the basis for defining the documents' internal representation within the OMS.

The aim of the evaluation reported here is to answer the question, whether object-oriented databases are suitable for storing abstract syntax-graphs that are produced in syntax-directed environments. In particular, we are interested, whether we can use the powerful data modelling capabilities, the versioning mechanisms, the concurrency control concepts and the functionality for data administration which they offer without loosing the performance we could gain from OMSs such as GRAS that are dedicated to syntax-directed tools.

Following the method we described in [ES92], we defined a benchmark that measures the behaviour of OMSs when they are used in syntax-directed design and specification environments. This benchmark is precisely defined in section 2. We have implemented this benchmark twice in order to evaluate GRAS and GemStone. In sections 3 and 4 we sketch the design decisions that led to the implementations of the benchmark. Section 5 aims at a comparison of the implementations and the results. A detailed description of the results we gained from performing the benchmark in different configurations.

2 The Merlin OMS Benchmark

For reasons that are beyond the scope of this report and extensively discussed in a companion paper [ES92], OMS benchmark definitions have to be abstract and to consist of the definition of (1) a conceptual OMS schema, (2) a number of benchmark operations and (3) one or more initial object base states. This section precisely defines the Merlin OMS benchmark. The definition was driven by the process model for OMS benchmark definitions that is also given in [ES92]. According to that reference, in a first step the scope of the benchmark has to be clarified by a kind of requirements analysis for a virtual software engineering application that uses an OMS. The results of the analysis are given in the first subsection. They are basically used for determining qualitative aspects of the benchmark definition such as the conceptual schema, or the benchmark operations. For also going on realistic assumptions concerning quantitative aspects of the benchmark, such as parameters of the operations or size and quantitative structure of the initial database, we present in the second subsection the results of a quantitative analysis we performed using existing documents of the previously identified types. Then the definition of the Merlin OMS benchmark components is performed, i.e. we present the conceptual schema of the benchmark in the third subsection, the definition of benchmark operations in the fourth subsection and the possible initial database states in the last subsection.

2.1 Requirements Analysis

The Merlin benchmark is intended to answer the question whether object-oriented database systems perform fast enough to be used within syntax-directed SDEs. To be able to define the benchmark appropriately, we have firstly to clarify the requirements we put on a particular SDE as some of them influence also the OMS used in the SDE.
As worked out in [ES92], these requirements are determined by (1) the document types that will be processed in the SDE and whose instances are stored in the OMS and (2) the consistency constraints that are applied to the documents and assured by the OMS.

2.1.1 Syntax Definition of Documents

For the Merlin benchmark we consider two highly integrated, syntax-directed design and specification tools each operating on one document type. The first document type allows for a graphical definition of modules and their import-relationships.
The second type is dedicated to the detailed definition of export- and import-interfaces of modules that were defined in the first document type. In particular, it allows the declaration of an export-interface by means of types, procedure heads and function heads. Procedures and functions, subsumed by the term *operation* in the following, have a name and an optional parameter list, that may consist of call by value and call by reference parameters. Moreover, they may be annotated with an optional comment. The import relationship is defined by a number of import lists. Each import list consists of a module name from which the module imports and a list of type and operation names that are imported. The concrete syntax of the two document types can be obtained from the grammars shown in figure 1.

```
<arch>          ::= ARCHITECTURE <arch_id>
                      <mod_list>
                    END <arch_id>.
<mod_list>      ::= <module> | <module> <mod_list>
<module>        ::= MODULE <mod_id>
                         <opt_comment>
                         <opt_imp_part>
                       END <mod_id>;
<opt_comment>   ::=  | <comment>
<opt_imp_part>  ::=  | <imp_part>
<imp_part>      ::= IMPORTS FROM: <imp_list>
<imp_list>      ::= <mod_id> | <mod_id>; <imp_list>
<mod_id>        ::= <ident>
<arch_id>       ::= <ident>

<ident>         ::= [A-Za-Z][A-Za-Z0-9_$]*
<comment>       ::= /"*"([^*/]|[^*]"/"|"*"[^/])*"*"/
```

```
<module>       ::= MODULE <mod_id>
                      <opt_comment>
                      <export_part>
                      <opt_imp_part>
                   END <mod_id>.
<opt_comment>  ::=    | <comment>
<export_part>  ::= EXPORT PART:
                     EXPORT TYPE: <typ_id>
                     OPERATIONS: <op_def_list>
<op_def_list>  ::= <op> | <op>;<op_def_list>
<op>           ::= <func> | <proc>
<func>         ::= FUNCTION <op_id><opt_par_list>:<typ_id>
                   <opt_comment>
<proc>         ::= PROCEDURE <op_id> <opt_par_list>
                   <opt_comment>
<opt_par_list> ::=    | (<par_list>)
<par_list>     ::= <par> | <part_list>
<par>          ::= <cbv> | <cbr>
<cbv>          ::= <par_id>:<type_id>
<cbr>          ::= VAR <par_id>:<type_id>
<opt_imp_part> ::=    | <import_part>
<import_part>  ::= IMPORT PART: <import_list>
<import_list>  ::= <import> | <import> <import_list>
<import>       ::= FROM <mod_id> IMPORT: <imp_obj_list> ;
<imp_obj_list> ::= <imp_obj> | <imp_obj>, <imp_obj_list>
<imp_obj>      ::= <typ_id> | <op_id>
<typ_id>       ::= <ident>
<mod_id>       ::= <ident>
<op_id>        ::= <ident>

<ident>        ::= [A-Za-z][A-Za-z0-9$_]*
<comment>      ::= /"*"([^*/]|[^*]"/"|"*"[^/])*"*"/
```

Figure 1: Syntax Definitions used for Merlin Benchmark

2.1.2 Definition of Consistency Constraints

The following consistency constraints are applied to documents of the previously defined types:

1. Modules that occur in an architecture are specified in detail in the specification language and vice versa.
2. Each import relationship in the architecture is specified in detail in the specification language and vice versa.
3. Names of modules, types and operations are unique within an architecture.
4. Modules that participate in import-relationships do exist.
5. Objects that are imported within one import-relationship are exported by the resp. module.
6. Cyclic import relationships are not allowed.
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.

The consistency constraints may only be violated as long as user-interactions are ongoing, i.e. user interactions can be viewed as transactions in the sense that they lead from one consistent document state to another.

2.2 Quantitative Analysis of existing Documents

To be able to define the parameters of benchmark operations as well as the proportions and the overall amount of objects representing different increments in the initial object base in a realistic way, we have to get an idea how documents of the previously defined types look like. We have therefore acquired existing design- and specification documents of an innovative software project for a detailed examination. In a first step, we had to transform the documents in a form that they were correct with respect to our grammars. We could achieve this by basically translating the keywords from German into English as the abstract syntax was nearly identical. Then we developed a program size metric that contains those aspects we were interested in. After that, we used `lex` and `yacc` to generate a parser that was capable of analysing the documents with respect to the metric. The metric as well as the analysis results can be obtained from table 1. They are based on some 5,000 lines of specification.

Metric component	Analysis result
Number of modules / architecture	18
Average number of exported types / module	0.3
Average number of exported procedures /module	11.0
Average number of exported functions / module	6.6
Average number of import lists / module	5.6
Average number of identifiers / module	148.8
Average number of comments / module	18.6
Average number of identifier / import list	6.0
Average number of call by value parameters / operation	1.7
Average number of call by reference parameters / operation	0.6
Average size of identifiers in architecture [Byte]	12
Average size of comments in architecture [Byte]	229

Table 1: Analysis results gained from specifications in an innovative project

2.3 Conceptual Schema modelling

In this step we define the internal data structures of the benchmark. We therefore start with modelling the internal data structures for the document types defined in subsection 1 in terms of an E/R diagram. This E/R diagram represents the conceptual scheme of an abstract syntax tree. In order to speed up consistency checks, this conceptual scheme is then enhanced with additional context-sensitive relationships. It then models the internal conceptual scheme of the class of possible abstract syntax-graphs. Finally, this scheme is simplified as described in [ES92] in order to reduce further development and implementation costs.

2.3.1 Systematic Derivation of E/R Model from Syntax Definition

As we only want to store abstract representations and derive the concrete ones using an unparsing mechanism, we transform in a first step the normalised EBNF into a tree grammar [DGKLM84]. This grammar provides the necessary information for deriving an E/R model that defines the document structure. The tree grammar defining the abstract syntax of the documents considered for the Merlin benchmark is depicted in figure 2.

To get a conceptual E/R model for an integrated syntax-tree, we can think about two approaches. The first is to integrate the tree grammars and to use the integrated grammar for the derivation of the E/R model. The second approach is to derive two E/R models each from one E/R diagram and after that to integrate the two E/R models. We choose the first approach, as most of the increments of the architectural language are also covered by increments in the specification language. We thus are allowed to simply add those operators and phyla of the tree grammar to the architectural language, that do not occur in the grammar of the

```
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 2: Abstract Syntax for Merlin Benchmark

module specification language. Then we can apply the following method to the resulting tree grammar.

First, we define an entity that represents the root of the syntax tree and carries the same name, as the root operator of the tree grammar. This entity is in our case the entity *arch*. To this initial E/R diagram, we apply for each operator and phylum of the grammar exactly one of the substitutions defined in table 2.

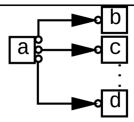
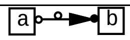
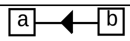
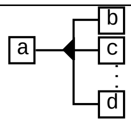
For tree grammar component of type	Substitute $\boxed{\text{a}}$ with
Fixed arity operator `a -> B C .. D`	
List operator `a -> B ...`	
Atomic operator `a::=B`	
Phyla `a::B C ... D`	

Table 2: Transformation of tree grammar into E/R diagrams

According to each fixed arity operator we allow to connect entities and relationships as fathers and sons in the syntax tree. Atomic operators which represent leaves of the syntax tree are defined as sub-entities that inherit their properties from predefined entities. An entity representing the left-hand-side of a list operator is connected by a one-to-many relationship with an entity that represents the right hand side of the list operator. Phyla that allow only one operator are replaced by that operator. Finally, Phyla that allow more than one operator become super-entities of an inheritance relation in which each sub-entity represents one operator. If the right hand side of an operator contains a phylum for which an entity has already been

included, we do not define it redundantly, but divert the relationship to the already existing entity.
In the sequel, we call the relationship that is drawn with solid lines here *aggregation relationship*.

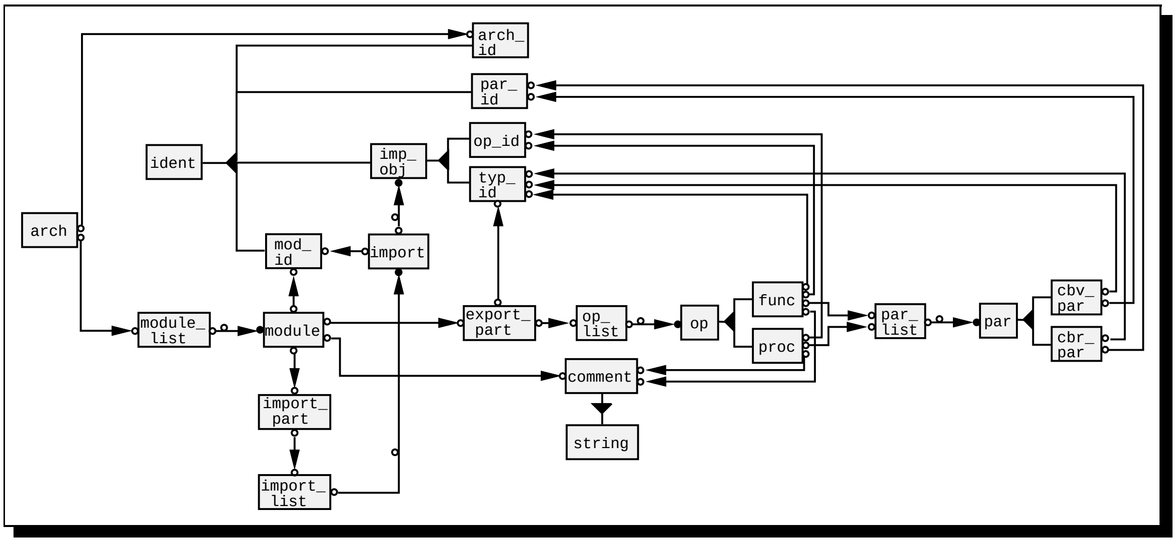


Figure 3: E/R diagram deduced from Grammar

When we applied this method to the grammars defined for the Merlin benchmark, we obtained the E/R diagram shown in figure 3. The initial E/R diagram contained only an entity type `arch`. To this diagram we applied the rule for fixed arity operators according to `arch -> ARCH_ID MOD_LIST`. As the phyla `ARCH_ID` and `MOD_LIST` allow only one operator each (`arch_id` and `mod_list`), we could include the entity types `arch_id` and `mod_list` that directly represent those operators and omit any representation of these phyla. Otherwise, e.g. in the case of the phylum `OP` that allows the two operators `func` and `proc`, we had to apply the rule for phyla and thus to include an entity type `op` as a super type of `func` and `proc`. The atomic operators, such as `comment` are supposed to inherit their properties from the predefined entities that represent the right hand side of the resp. atomic operator. In case of `comment` this would be a predefined entity `string`. List operators such as `op_list` are represented by ordered one-to-many relationships that connect the entity representing the left-hand-side (`op_list`) and the right-hand-side (`op`).

2.3.2 Introduction of context sensitive Relationships

So far, we have not considered the existence of the consistency constraints defined within the requirements analysis subtask. This subsection describes how consistency checks may be accelerated by additional relationships introduced to the conceptual model of the object base, thus transforming the modelled abstract syntax tree into an abstract syntax graph.

The result of this activity for the Merlin benchmark is the E/R diagram shown in figure 4. The first 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.
The first dictionary contains for each architecture references to all identifiers that are declared somewhere in the architecture. The second dictionary contains for each module references to the identifiers that are exported by the module, i.e. that may be used as imported objects.

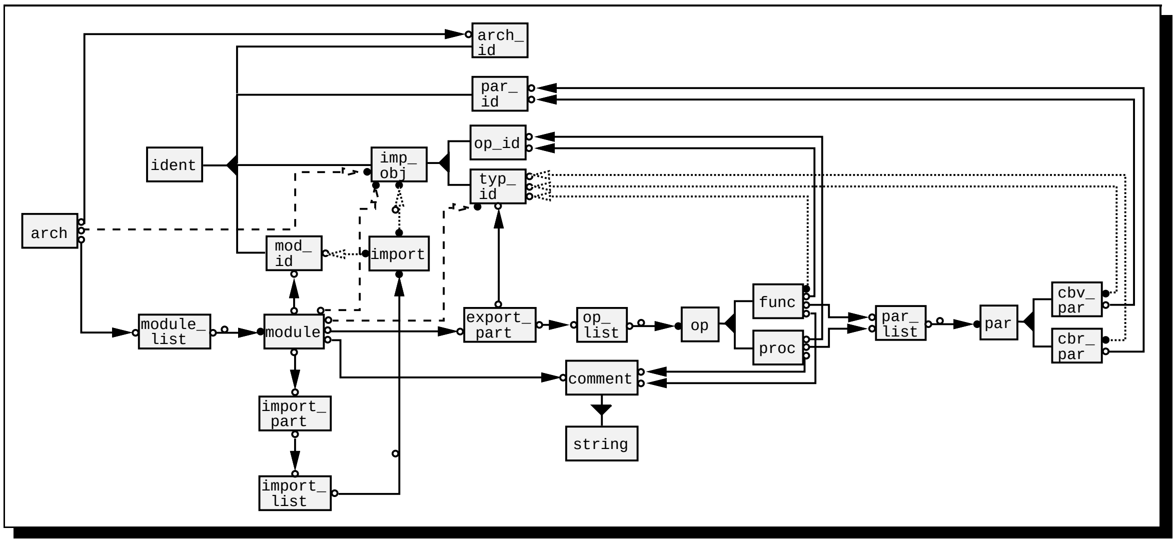


Figure 4: E/R Model enhanced with context sensitive Relationships

The last dictionary contains for each module references to those type identifiers that may be used within the module. This allows to answer the question whether an identifier has already been declared in the scope of an architecture by a single query to the respective dictionary.
To ensure that inconsistencies are not introduced due to modifications of objects, we transformed some of the aggregation relationships into reference relationships. 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, in import lists, the copies of identifiers of imported modules and objects are transformed into references to the resp. identifiers. This not only allows omitting of change propagations, but also enables checks whether an exported type or operation is actually used.

2.3.3 Schema Simplification

Up to now, the model of the conceptual schema of the benchmark and the model of a future software engineering application are identical. Moreover, at this state of the process the knowledge needed for permissible simplifications of the benchmark is accumulated in the E/R diagram. Hence, it is down to this activity to distinguish the schema of the application from the one of the benchmark in the sense that the conceptual schema of the benchmark is significantly simplified. We identify the possible points for simplification:

1. Relationships that start from or lead to all sub-entities of an inheritance relationship can be replaced by one relationship that starts from or leads to the super-entity.

2. Entities that do not participate in any relationship except that they are end of an aggregation type relationship and that are not editable (c.f. subsection 2.1) can be transformed into attributes of the entities, where the aggregation relationship starts.

3. Sub-entities of an inheritance relation, that participate in the same relationships and carry the same attributes as another entity of that inheritance relation may be 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 can be removed together with its sub-entity. The super-entity takes over all relationships, the sub-entity participated in, as well as all attributes it carried.

5. An entity that neither participates in a context-sensitive or inheritance relationship nor carries attributes and is source of only one aggregation relationship can be 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 applied repeatedly in mutual exclusion and whereas rule 5 may be applied repeatedly only if the application of rules 1–4 is finished.

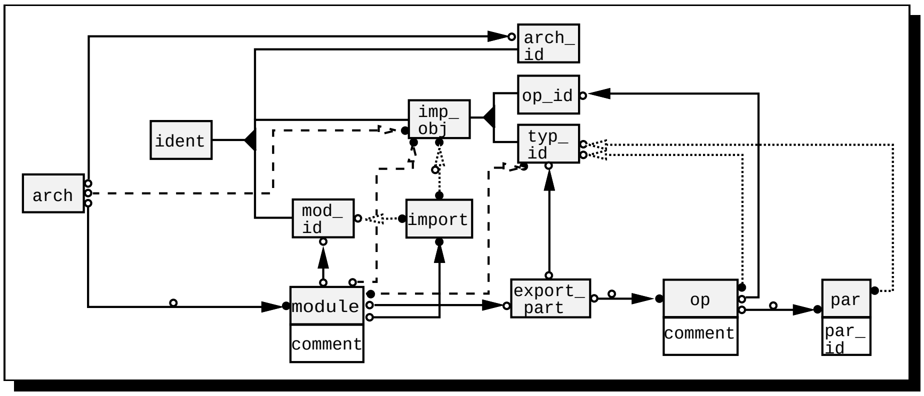


Figure 5: E/R Model defining Merlin benchmark schema

Using these simplifications we are able to simplify the E/R diagram shown in figure 4. The result of this process is depicted in figure 5. 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.

At this stage, the E/R modelling phase of the benchmark definition is finished and figure 5 defines the conceptual schema of the Merlin benchmark. Based on this schema we can now define the benchmark operations.

2.4 Definition of initial Object Base States

We now have to define the possible initial object base states of the benchmark. The reason for not operating on an empty object base is twofold. Firstly, the amount of data stored in an

OMS has an impact on the performance, since the OMS could keep a small amount of data in buffers and at least speed-up read accesses, which it could not with a large amount. Secondly, consistency constraints may be easier to assure with a small initial object base than with a large one. The size of the initial object base should possibly be varied to measure the influence of these two aspects.
To go on realistic assumptions not only for the size, but also for the structure of the data base, we should use the data produced during the quantitative analysis phase. Unfortunately, we can not interpret it in an one-to-one manner, as we simplified the data structures of the benchmark. Thus the metric has to be adjusted to the new situation in the following way: We combine some entity types that formerly represented different syntactical increments with their super-entity types. Consequently, each entry of the metric that is no longer represented by an entity type in the E/R diagram counts for the respective super-entity type.

The structure of the modules used for the creation of the initial object base in the Merlin benchmark is closely related to the measurement results given in table 1. It is identical for each module. The contents of a module (e.g. the names of procedures or modules) may vary but should have exactly the same length as in the other modules. Table 3 shows the metric values for such a module.
As we have combined the entity types `cbv_par` and `cbr_par` in `par`, we summed their metric entries up to obtain the number of parameters per operation. Similarly we added the entries of functions and procedures to derive the number of exported operations.

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 3: Metric for a module in the initial object base

The amount of data contained in the initial object base is kept increasable in order to measure the influence of the size of the object base on the performance of OMS operations. This also allows us to check whether there are any (not documented) OMS specific restrictions on the object base's size.
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} ($j \in \{1, \ldots, n-2\}$). Figure 6 shows as an example the architecture in the initial object base with 4 levels.

2.5 Definition of Benchmark Operations

We finally have to define the benchmark operations. Therefore we define the operations of the Merlin benchmark as if they would implement user-interactions to documents. Measuring the execution times of these operations will then allow us to forecast the OMS specific execution

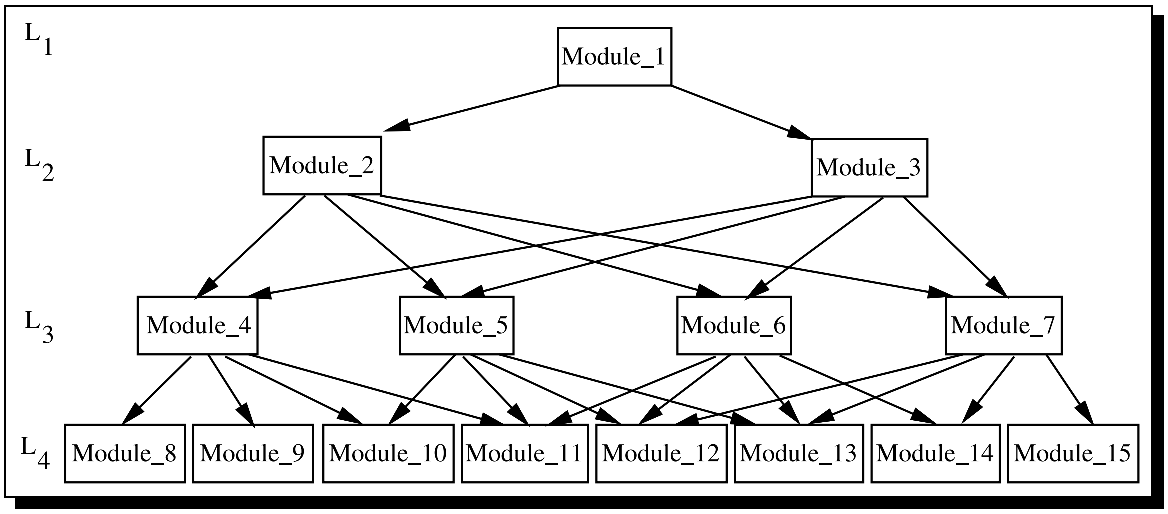


Figure 6: Initial Object Base with 4 Levels

times a software engineering application's user-interaction would have when built on top of the OMS under investigation.
The operations of the Merlin benchmark are defined as if they would implement user-operations that are made persistent immediately and perform necessary change propagations in order to assure the invariance of the previously defined consistency constraints.
The operations preserve the quantitative structure of the initial database, because the operation parameters are chosen according to table 6.

The operations can be clustered into four groups. Operations of the first group create increments of the two document types like modules, types, operations, parameters, and comments. The second group is dedicated to measure the impact of changes to these increments. The third group simulates tool operations that perform analysis. Finally the last group deletes all previously created objects.

OpenOMS Open the access to the OMS, i.e. perform all necessary operations such as authorisation or reservation of buffers to access data in the OMS.

CreModul Create n new modules in the architecture. Set their names to `Module_No_j+2`^{i}, ($j \in \{0, \ldots, n-1\}, i =$ levels in the initial objectbase). Set the graphical coordinates to $(50 \cdot j, 100)$.

CrModTyp For each created Module set the export type identifier to `Type_No_j+2`^{i}, ($j \in \{0, \ldots, n-1\}, i =$ levels in the initial objectbase).

CrModCom For each module created by **CreModul** expand the module comment increment to an arbitrary string which is 256 bytes long.

CrModImp For each created module create a number of four import relationships to arbitrary given modules of the bottom most level of the initial objectbase.

CrImpObj For each created import relationship of each created module increment expand the list of imported objects to the exported type and the $1^{st}, 3^{rd}, 7^{th}, 11^{th}$, and 14^{th} operation of the respective module.

CrExpOpe Create a list of 17 operation increments in each created module, update their name to `Mod`j`_Func_`k with j being the number of the module as determined by the **CreModul** operation and $k \in \{1, \ldots, 17\}$ and expand their type to the type which is exported by the module.

CrOpePar Expand each operation parameter increment in each created module to a parameter list with three call by reference parameters. Expand the parameter names to arbitrary unique identifiers (12 bytes) and the parameter types to the types which are imported by the first three import lists of the module.

CrOpeCom For each operation expand the operation comment increment to an arbitrary string of length 256 bytes.

UnparMod Create textual representations of the created modules in a string of the host programming language.

UnparArc Create a textual representation of the architecture in a string of the host programming language.

AnaUsage For each module of the bottom most level of the initial objectbase return the names of modules to which it exports objects. Return the names as a list of the host programming language.

ClosTrav Return the names of those modules, which are reachable from the module at level 1. Return the names as a list of the host programming language.

ChModNam Change the module name of each module to another unique name with the same length and update the name in the import parts of those modules which use the module.

ChModTyp Change the exported type of each module to another name with the same length and update the name in the import parts and parameter lists of those modules which use the module.

ChModCom Change the contents of the module comment of each created module to an arbitrary other string of the same (256 bytes) length.

ChOpeNam Change the names of the exported operations of each module to another name with the same length and update the name in each import list which imports the operation.

ChParNam Change the names of each operation parameter to other values with the same length.

DlOpeCom Delete every operation comment in all modules that has been created during the benchmark.

DlOpePar Delete each operation parameter in each module that has been created during the benchmark.

DlOperat Delete all exported operations in all modules that have been created during the benchmark.

DlImpObj Delete all imported objects in all modules that have been created during the benchmark.

DlImpRel Delete all import relations in all modules that have been created during the benchmark.

DlModule Delete all modules that have been created during the benchmark.

CloseOMS Perform all operations that are necessary to start the benchmark again.

The operations must be performed in the order in which they appear in the list given above, i.e. The benchmark should first create n new modules, then expand for each newly created module the exported type and so on.

3 Using GRAS

The document management system GRAS [LS88] has been developed at University of Osnabrück as part of the IPSEN project. The overall goal of GRAS is to provide support for efficient storage of documents like architectural specifications, module specifications or source code in an incremental and highly integrated SEE.

The conceptual data model in GRAS is based on directed, attributed graphs. These graphs allow to persistently implement abstract syntax graphs of documents.

A database in GRAS is called a graph-pool. It consists of an arbitrary number of graphs. GRAS does not provide any means for a schema definition of graph-pools. The only entity type is `node` whereas the only relationship type is `edge`. For the implementation of graph access and manipulation operations GRAS provides programming interfaces to Modula-2 and C. Despite programming an application, there are no other means to inspect or modify a graph-pool. In particular, browsers and administration tools are not available.

The reason why we consider GRAS for an implementation of the Merlin benchmark is that it has been explicitly dedicated to *efficient* storage of abstract syntax graphs. Thus, it provides a suitable yardstick for other database systems.

3.1 Storing the documents of the Merlin Benchmark in GRAS

This subsection explains how we stored the documents of the Merlin Benchmark in GRAS. Therefore, we first present the basic ideas of the programming interfaces to GRAS. Then we discuss some limitations of GRAS that have an impact on GRAS' usage. Finally, we show how we mapped abstract syntax graphs as defined in figure 5 on attributed graphs stored in GRAS.

3.1.1 The Programming Interfaces to GRAS

This subsection roughly sketches the basic operations that span up the programming interfaces to the GRAS system.

Basically, these operations can be subsumed as follows:

- graph-pool operations,
- graph operations,
- partial match query operations (PMQs).

A graph-pool consists of a set of graphs that are logically related. The programming interfaces offer operations to create or delete graph-pools and to create or delete graphs within a graph-pool. Moreover, they provide operations to open and close a graph identified by a unique user-defined graph name. This is necessary in order to access the contents of a graph and to make any modifications applied to the graph become persistent, respectively. The programming interfaces therefore offer means for assigning and removing external names to or from nodes as well as for searching nodes that are identified by an external name.

A graph consists of a finite set of labeled nodes that may be attributed. These nodes are connected by labeled, directed edges. To access and manipulate these graphs, the programming interfaces provide a number of further operations:

- Creation and deletion of labeled nodes,
- Creation and deletion of labeled edges,
- Assignment and retrieval of attributes to/from nodes.

During creation of a node, GRAS assigns a unique object identifier to a node which may in turn be used for efficiently accessing the node. Besides unique object identifiers, applications can define unique external names to nodes. These names enable associative searches for nodes in a graph.

GRAS provides means to perform partial match queries in order to implement graph traversals. A partial match query is specified by two components. Having specified a node identifier and an edge label, an application can retrieve a single node or a set of nodes that are connected to or from the node by an edge with the given label.

3.1.2 Limitations of GRAS

There are a number of limitations when using GRAS which heavily influence the design of an SEE in general and that of the Merlin benchmark in particular. In order to better understand this design, we now discuss the mentioned limitations.

Number of Nodes in Graphs The number of nodes in a graph is limited to 2^{16}. The overall amount of nodes in the syntax graph representation of documents that are being produced in small project is already beyond that number. Thus, we are forced to distribute the syntax graph over a number of graphs.

Open Graphs In order to achieve efficient caching of nodes and edges, the designers of GRAS have decided to require explicit opening and closing of graphs from an application. Only graphs that are opened may be accessed and modified. The number of graphs that may be open at once, however, is limited to 20. Hence we can not keep all graphs open at any time.

Size of Attributes The upper bound for an attribute's size is 251 Bytes. Hence, if we still want to store longer attributes in GRAS, we would be forced to split them artificially and distribute the parts over a number of nodes.

Transactions GRAS provides a mechanism to group calls to its programming interface as transactions. These transactions may be nested. Unfortunately transactions in GRAS apply to exactly one graph, i.e. operations that perform accesses and changes in more than one graph can not be performed as transactions.

External Edges GRAS does not provide direct support for edges that span over two different graphs. If we want to implement such edges, we have to store attributes at the source node containing the object identifier and the graphname denoting the node they lead to. A similar approach is required for the reverse edge. As a matter of fact, we therefore have to open the graph to which the edge leads and close it afterwards. This approach, however, does not assure that referential integrity is preserved.

3.1.3 Implementing the Merlin Benchmark Schema with attributed Graphs

This subsection defines how entities defined in the benchmark schema depicted in figure 5 are represented in terms of nodes and edges of graphs managed by GRAS. Therefore, we first define how we structured the database in terms of graphs.

On account of the limited number of nodes that may be contained in a graph, we decided to split the initial database into a number of graphs. These graphs are stored in a graph-pool called *InitialDatabase*.
It contains a graph in which entities of types `arch`, `module`, `ident` and `import` are implemented. This graph is called *system-graph* in the following. During a benchmark run, this graph will always be kept open. Besides the system-graph, we implemented one graph for each module of the initial database. These graphs are called *module-graphs* in the rest of this section and they are only opened on demand.

SystemGraph The system-graph contains all information required to represent the architecture of the modules in the initial database. It is designed, as if it had to serve as a data-structure of a graphical editor allowing architectural design. Figure 7 depicts an example representation of such a module hierarchy.

Boxes in that figure represent nodes, whereas arrows represent edges. Edges depicted by solid arrows implement aggregation type relationships whereas those drawn by dashed arrows implement reference type relationships. Moreover, node labels and edge labels are shown within the boxes and near the arrows, respectively. Finally, a number depicted in the upper

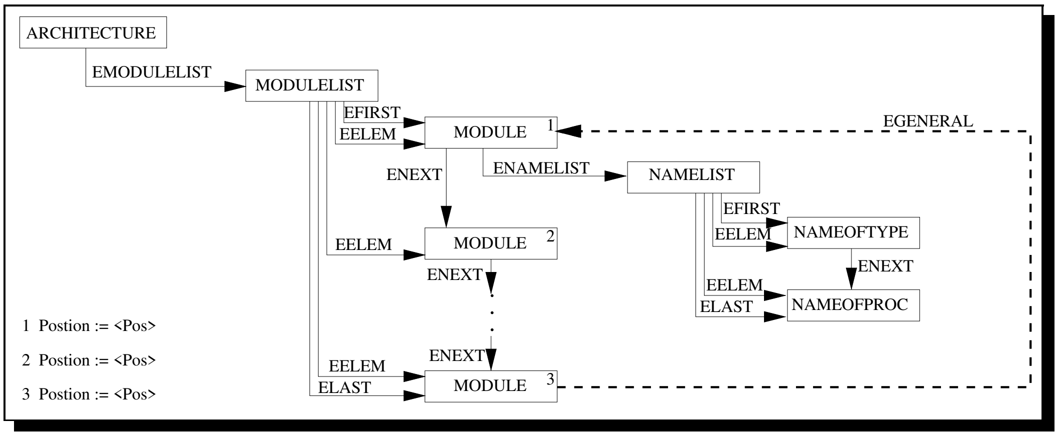


Figure 7: System-Graph

right corner of a box depicts the value of a node attribute as indicated by the assignment statements in the lower left corner of the figure.

The entity type `arch` in figure 5 is implemented by nodes labeled with *ARCHITECTURE*. GRAS does not provide any direct means to implement ordered 1:n relationships such as the relationship in figure 5 starting from `arch` and leading to `module`. These relationship have to be implemented by additional nodes and edges. To implement the previously mentioned relationship, for instance, we introduced a new node labeled with *MODULELIST* as anchor of the list and new edges labeled with *EFIRST, EELEM, ENEXT* and *ELAST*. Arbitrary many nodes labeled with *MODULE* can be connected to this anchor by edges labeled with *EELEM*. An order of these elements is defined by edges of type *ENEXT*. Edges labeled with *EFIRST* and *ELAST* are devoted to efficiently find the first respectively last element of the list. Nodes labeled with *MODULE* have attributes that are intended to store the graphical coordinates of a module in a graphical architecture depiction. The identifier of a module is not implemented as a node, but as an external name assigned to the respectively node.

The dictionary type relationship from `arch` to `module` must be queried associatively in order to search for given identifier values. As the number of identifiers which participate in this relationship becomes very large in a big initial database and as the query will be used often, the query must be implemented very efficiently. GRAS' external names provide means to implement associative access to nodes very efficiently based on external hashing. Unfortunately, the scope for these queries are the external names defined in **one** graph and identifier nodes are distributed over all module graphs. Searching for the names in all module-graphs is infeasible because the time needed only to open and close a graph was found in [DHK+90] to be much more than one second. Thus, we have decided to store nodes that represent identifiers redundantly: one in the module graph representing the module where the identifier is declared and another one in the system-graph. In the latter the identifier node is annotated with an external name which is in turn the basis for the above mentioned queries.
To implement this strategy, we added to each node labeled with *MODULE* a node of type *NAMELIST* which carries copies of the identifier nodes declared in the respective module.

The import relationships between modules are represented in the system-graph by edges labeled with *EGENERAL*. They are refined in the module-graph.

ModulGraph Figure 8 gives an example for the structure of a module-graph following the same notation used for the system-graph. In spite of its complex structure, only the most important components are depicted there.

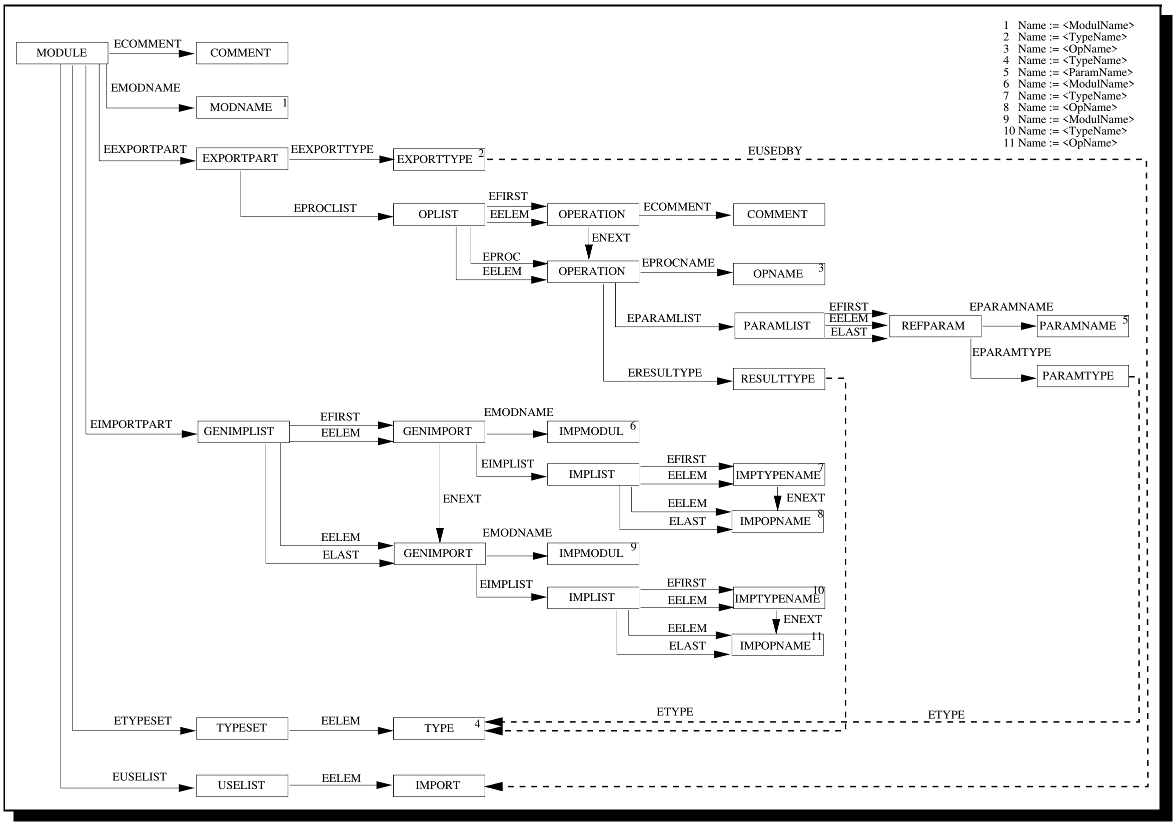


Figure 8: Modul-Graph

The entity types `module, mod_id, export_part, op, par` and `import` defined in the benchmark schema are implemented in a straight-forward manner by the nodes with the label *MODULE, MODNAME, EXPORTPART, OPERATION, PAR* and *GENIMPORT*. 1:1 aggregation type relationships in the benchmark schema are implemented by edges, whereas ordered 1:n aggregation type relationships are implemented by lists (*OPLIST, PARLIST* and *GENIMPLIST*) in the way explained above.

Entities of types `module` and `op` in the benchmark schema have an attribute denoted as `comment`. As comments of modules and operations are often larger than 251 Bytes (c.f. the analysis results in table 1), they can not be stored as attributes of nodes. Thus, we decided to store comments in the Unix File-System and to maintain only placeholder nodes labeled with *COMMENT* in the module-graph. The comments of a module are stored in a single Unix directory. The file-name of a comment's file is defined by the concatenation of the directory's name with the node-number of the comment node.

All reference type relationships defined in figure 5 can not be implemented by simple edges, as the nodes they would have to lead to are stored in other graphs. That would require edges spanning over different graphs which are not supported by GRAS. To still be able to express these relationships, we have to introduce redundant nodes for each entity, instances of these relationships lead to. In particular, we include nodes labeled *IMPTYPENAME* and *IMPOP-NAME* to implement the relationship to entities of type `imp_obj`, nodes labeled *IMPMODULE* to implement the relationship to entities of type `mod_id` and nodes labeled *TYPE* to implement the relationship to entities of type `type_id`. Note, that each of these nodes is redundant to at least one other node. This redundancy has, as we will see in section 5.2, a great impact on the performance of benchmark operations.

That redundancy also complicates maintenance of consistency constraints, as additional change propagations are required now. Consider the change of the name of an exported type or operation which is imported elsewhere. In order to meet the consistency constraint, that each imported type or operation is exported by the imported module, we have to propagate the changed name also to all places, where it is being used. To accelerate the time required for searching these places, we maintain a *USELIST* which contains nodes representing those modules that use resources from a module. Each export being used in some module, is connected by an edge labeled with *EUSEDBY* to a node in the *USELIST*. During change propagation we can follow these edges to find the related module-graphs in which we have to update the import interface.

The dictionary type relationships starting from entities of type `module` are implemented in different ways: The first relationship which leads to entities of type `imp_obj` and is intended to define those names, that are exported by the module is defined by external names. The second relationship intended to contain all type identifiers valid in a module can not be implemented by external names of the module-graph, as these are already being used for the first dictionary type. Thus, we store one node labeled *TYPESET* in each module-graph which has edges to each node in the graph representing a valid type identifier.

3.2 Architecture of the Benchmark

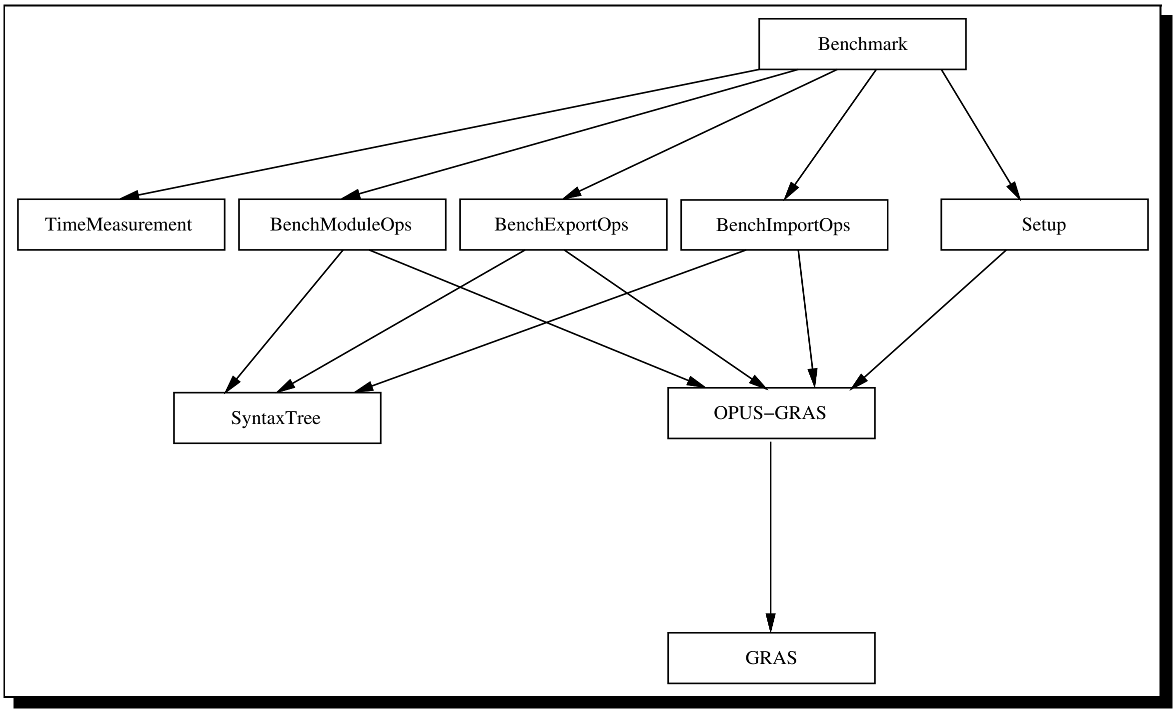


Figure 9: Architecture of the Merlin Benchmark implemented on top of GRAS

Figure 9 shows an overview of the architecture we used for implementing the Merlin benchmark on top of GRAS. In this figure rectangles represent modules and arrows describe the usage-relationship.

The GRAS module represents the programming interface to the GRAS system. It provides the operations to manipulate graph-pools and graphs and to perform the PMQs previously described in this chapter.

The OPUS-GRAS module is composed of several modules. It contains modules for each increment which encapsulate operations to expand, change, analyse and delete subincrements. Besides there is one module to manage the use-list of a module. These modules all use the operations of the programming interface.

The module *SyntaxTree* is devoted to manage temporary storage of object identifiers that represent the increments of different documents in a syntax tree like structure.
It provides means for insertion and deletion of new identifiers into the tree and for a variety of different traversals.

The modules *BenchModuleOps*, *BenchExportOps* and *BenchImportOps* implement the benchmark operations as defined in subsection 2.5. Therefore they call respective operations of modules in *OPUS-GRAS* and use *SyntaxTree* to temporarily store object identifiers that are returned by the programming interface.

The module *Benchmark* implements the main program. It can be invoked from the Unix command line. It allows to pass an integer n indicating how many modules shall be created during the benchmark run. *Benchmark* uses *TimeMeasurement* in order to measure the elapsed real time between start and end of a benchmark operation. *TimeMeasurement* logs these figures into a text file that may be used for the result analysis. Besides that, it uses the module *Setup* which provides means for database login and logout.

3.3 Performance Measurement Conditions

We implemented the benchmark with GRAS Version 4.14 using the C programming interface. The detailed configurations of the machine we used can be obtained from Table 4.

Machine	Sun SparcStation IPX
Operating System	SunOS 4.2 Release 4.1.1
Main Memory	40 MB
Disk	WRENV IV 94181-385h with 320 MB
Disk controller	Emulex MD 21

Table 4: Description of the system which performed the benchmark with GRAS

For comparison purposes, we performed the benchmark when the machine was in multiuser mode. Other databases require remote procedure call facilities of the operating system to be available which are stopped in single user-mode. Besides the benchmark there was no load on the machine.

4 Using GemStone

GemStone [BMO+89] was the first fully object-oriented database system. Its development started about 10 years ago at Oregon Postgraduate Center, Beaverton, OR [CM84].

A GemStone schema is defined by a number of classes. These classes not only define structural properties of objects, but also objects' behaviour in terms of methods. Methods provide the only interface for the manipulation of objects of that class. GemStone provides a language called *OPAL* dedicated to class definition purposes.

GemStone comes with a number of initial classes called *kernel classes*. These include atomic classes like `boolean`, `integer`, `fraction`, `float` and `string` as well as object constructing classes like `list`, `set`, `bag` and a variety of different dictionaries. Application specific classes are built by inheriting from one of these initial classes.

GemStone provides two different transaction management strategies: *optimistic* and *pessimistic* transactions. In optimistic mode, transactions do not lock objects and GemStone detects conflicts between concurrent transactions at commit time. In pessimistic mode, conflicts are already detected at the time locks are acquired and transactions are requested to wait until they get the lock granted or to abort.

GemStone allows for distributed access to its databases by a client/server architecture. Therefore it runs a database monitor process called *stone*. This process performs physical access to the databases, controls user logins and preserves the integrity of databases against hardware and software failures by maintaining a log. In addition to the stone, the host runs for each application one other process called *gem* which performs the execution of methods by calling the *stone* using RPC.
An application may use two different versions of the programming interface in order to connect to its corresponding gem process. In *linked mode*, application and gem process are melt together and executed as one process on the machine that also runs the stone process. In *remote mode*, the application and its gem process are two different processes that may run on different machines and communicate by RPC.

GemStone offers a programming environment with a textual data browser called *TOPAZ*. TOPAZ also provides support for the database administrator as it is capable to perform backup procedures, user administration and basic performance measurement.

4.1 GemStone Classes for Merlin Benchmark

This subsection describes how we have implemented the conceptual schema of the Merlin benchmark in GemStone. Therefore, we first discuss how we mapped the E/R model in figure 5 to data structures contained in GemStone classes. Then we describe how we managed encapsulation of the data structures by methods. These methods are called by the benchmark to implement benchmark operations.

4.1.1 Setting up Classes for the Merlin Benchmark

Figure 10 depicts the classes we used for the Merlin Benchmark and shows how they are related to GemStone's kernel classes.

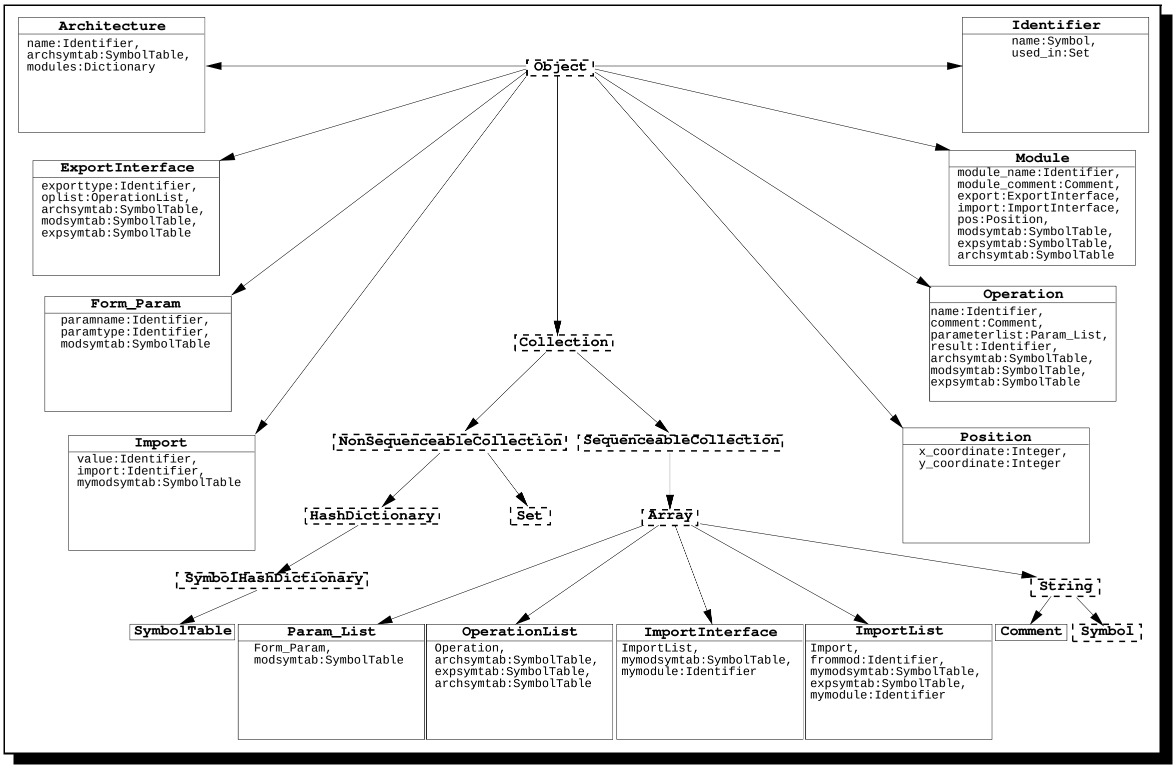


Figure 10: Classes contained in the Schema of the Merlin Benchmark

Rectangles represent classes whereas arrows denote the subclass-of relationship. Names of classes are typeset in bold at rectangles' top edges. Classes that belong to GemStone's kernel classes are represented by dashed rectangles whilst those classes added for the Merlin benchmark (called benchmark classes in the following) are depicted as solid rectangles. Instance variables of benchmark classes not being inherited from kernel classes are given in the lower part of the rectangles by their names and types.

For each entity type defined in figure 5 there is one benchmark class that implements this entity type. Attributes of entity types are implemented as instance variables of the respective classes.
As object-oriented databases do not provide an explicit notion of relationships, relationships are either implemented as instance variables or additional classes: relationships of cardinality 1:1 are implemented as instance variables whereas relationships of cardinality 1:n are implemented as sets or lists.

Consider as an example the entity type `module` in figure 5. It is implemented by class `Module`. The 1:1 relationship to entities of type `mod_id` is implemented by the instance variable `module_name`. The 1:n relationship to entities of type `import` is implemented by an instance variable of class `ImportInterface`. This class inherits the property of having arbitrary many unnamed instance variables from the kernel class `Array`. The class of these variables is constrained to be of `ImportList`. Additionally, `ImportInterface` declares two named instance variables that have no counterpart in figure 5 but are added for efficiency purposes: Firstly, mymodule implements a reference to the identifier of the module in which ImportInterface is contained. Secondly, mymodsymtab is a reference to a SymbolTable that contains those types that have been declared in the scope of a module.

The class SymbolTable implements relationships of type Dictionary. SymbolTable is defined as subclass of the kernel class SymbolHashDictionary. It provides efficient associative access to identifiers using their values. Instances of this class are used for several purposes.
Firstly, each instance of class Architecture has got a symboltable archsymtab that contains those identifiers declared within the scope of an architecture. If a new identifier is declared or the value of an identifier is changed, we use this symboltable to decide whether the new value is unique. Consequently, all classes (e.g. Module, ExportInterface, Operation) that declare new identifiers maintain a reference to the archsymtab of their architecture.
Secondly, each instance of class Module declares a symboltable expsymtab that defines those identifiers that are exported by that module. This symboltable provides means to efficiently decide whether or not a particular identifier is exported by the module. Moreover, it allows to efficiently locate the identifier. References to this symboltable are maintained by all classes that contribute to the export interface of the module (e.g. ExportInterface, OperationList and Operation) as well as those classes that use an export like e.g. ImportList.
Finally, each instance of class Module defines a symboltable modsymtab, that defines those types that are declared within the module. It is used to decide whether a type used as result type or parameter type in operations has been declared either in the export interface or by import of a type.

In order to implement the reference relationship of cardinality 1:n in which identifiers participate (c.f. figure 5) we added the instance variable used_in to class Identifier. Here we take into account that we have to navigate along this relationship in both directions. The implementation of benchmark operations assures, that the set used_in always contains those instances which refer to the identifier.

4.1.2 Method Definitions

The data structures defined for the benchmark classes as indicated by figure 10 are encapsulated by methods. These method provide the only means to modify instance variables of classes. GemStone distinguishes between classmethods and instancemethods. A classmethod is invoked when sending a message to a class, whereas an instancemethod is executed after a message has been sent to an instance. Both kinds of methods may be parameterised and return a result.

We use classmethods in benchmark classes for instance creation and initialisation purposes. That is, each class has one classmethod that creates on demand new instances of that class and initialises its instance variables.
Class Module, for example, has a classmethod CreateModule that has parameters of the types SymbolTable, Point and String. During execution it first of all creates a new instance of class Module. Then it initialises the module's instance variable archsymtab to the symboltable that was passed as parameter and initialises the instance variables expsymtab and modsymtab with two new symboltables which it creates. Then it initialises the module's instance variable pos with the position that was passed as parameter. After that, it creates a new identifier, sets its value to the string that was passed as parameter and initialises the instance variable

module_name to this identifier. Having finished that, it creates a new instance of class Comment and uses it for initialisation of module_comment. Finally, it creates new instances of classes ExportInterface and ImportInterface and initialises the instance variables export and import respectively.

Instancemethods are the baseline for the implementation of the benchmark operations defined in subsection 2.5. They are implemented in such a way that they assure the consistency constraints required in subsection 2.1.2.
For each class that directly implements an increment, we defined instancemethods that allow for expansion, changes and deletion of subincrements. These methods return boolean values that indicate whether or not the execution was successful. An execution may fail if it would violate a consistency constraint.

```
ExpandOperationName: aString

(ArchSymTab IsInSymbolTable: aString)
ifFalse:[
    OperationName := Identifier CreateIdentifier.
    OperationName SetIdentifier: aString.
    ArchSymTab EnterToSymbolTable: aString
            DefinedIn: OperationName.
    ExpSymTab EnterToSymbolTable: aString
            DefinedIn: OperationName.
    ^TRUE]
ifTrue:[^FALSE]
```

Figure 11: Example for Instancemethod in Merlin Benchmark Class

Consider, for example, the method ExpandOperationName defined in class Operation shown in figure 11 using the syntax of OPAL. It expands the name of an operation to a string passed as parameter. Therefore it first queries the symboltable of the architecture that contains all known identifiers, whether the given name is unique. If so, it creates a new identifier, sets its value to the given string and includes it into the symboltable of known identifiers as well as into the symboltable that contains the identifiers exported by the module. Then it returns TRUE in order to indicate a successful execution. In case the given string is found in the architecture symboltable, it returns FALSE to indicate that the execution is not completed successfully as a consistency constraint is violated.

4.2 Architecture of the Benchmark

Figure 12 shows an overview of the architecture we used for implementing the Merlin benchmark on top of GemStone. In this figure rectangles represent modules and arrows describe the usage-relationship.

The *GCI* module represents GemStone's C Interface. It provides operations for sending messages to objects. The object a message is inteded for is denoted by an unique object identifier which has to be passed as parameter. Besides that, GCI provides operations for transaction management (e.g. commits or aborts of transactions), session control (e.g. logging into and logging out of a GemStone session), and conversion routines between elementary C types and atomic GemStone classes.

The *GemStoneInterface* must be considered rather as a subsystem of modules than a single

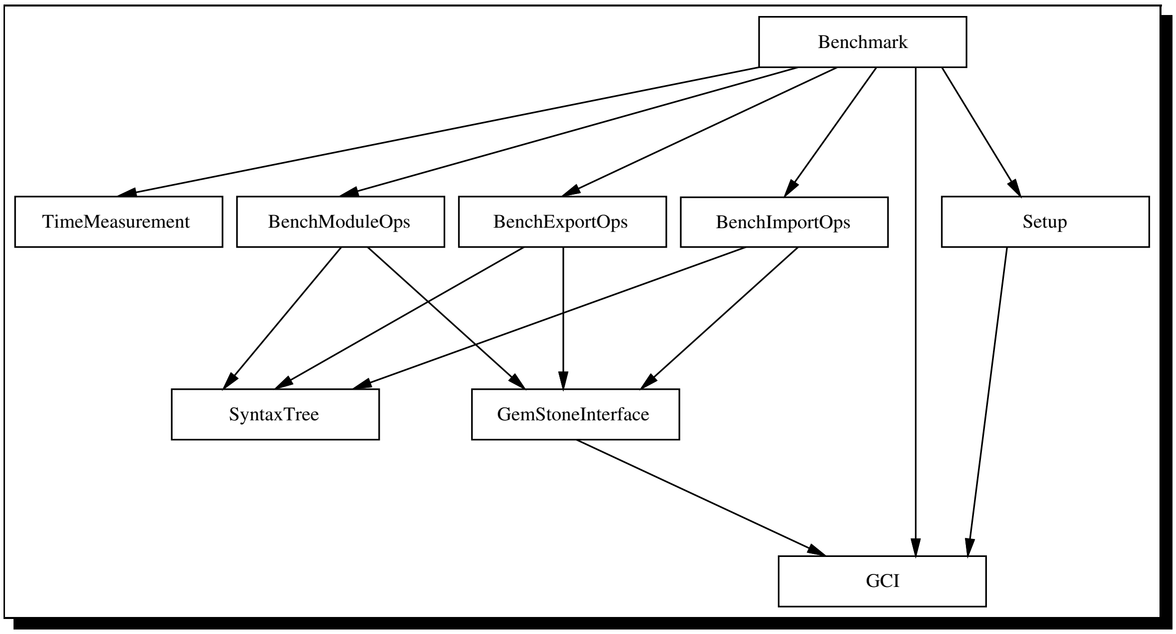


Figure 12: Architecture of the Merlin Benchmark implemented on top of GemStone

module. It contains modules for each increment which encapsulate operations to expand, change, analyse and delete subincrements.
Operations in these modules use the *GCI* to send messages to objects stored in GemStone. The interpretation of these messages by GemStone then cause execution of the methods sketched in subsection 4.1.2.
Parameters of these functions are always unique object identifiers that denote objects stored persistently in GemStone which shall receive the message. Additionally some functions have parameters that contain values of identifiers or comments, or analysis results.

The module *SyntaxTree* is devoted to manage temporary storage of object identifiers that represent the increments of different documents in a syntax tree like structure.
It provides means for insertion and deletion of new identifiers into the tree and for a variety of different traversals.

The modules *BenchModuleOps*, *BenchExportOps* and *BenchImportOps* implement the benchmark operations as defined in subsection 2.5. Therefore they call respective operations of modules in *GemStoneInterface* and use *SyntaxTree* to temporary store object identifiers that are returned by the *GemStoneInterface*.

The module *Benchmark* implements the main program. It can be invoked from the Unix command line. It allows to pass an integer n indicating how many modules shall be created during the benchmark run. Every benchmark operation is executed as one GemStone transaction using the optimistic mode. *Benchmark* uses *TimeMeasurement* in order to measure the elapsed real time between start and end of a benchmark operation. *TimeMeasurement* logs these figures into a text file that may be used for the result analysis. Besides that, it uses the module *Setup* which provides means for database login and logout. Moreover, it uses transaction management routines from the *GCI*.

Altogether, the implementation contains about 7,000 lines of C code heavily annotated with comments.

4.3 Performance Measurement Conditions

We implemented the benchmark with GemStone Version 2.5 using GemStone's C Interface Version 2.5. We ran the benchmark in two different hardware configurations. In the first configuration which we call *local mode*, the complete benchmark was executed on a SPARCstation IPX that also ran GemStone's stone process. Moreover the benchmark and GemStone's gem-process were linked into one process.
In the second configuration called *remote mode* the benchmark was executed on a SPARC-station SLC accessing GemStone's gem-process via RPC. The gem-processes as well as the database server process were executed on the same machine as in the local mode. The detailed configurations of the machines can be obtained from table 5.

	Local Mode	Remote Mode
Machine	Sun SPARCstation IPX	Sun SPARCstation SLC
Operating System	SunOS 4.2 Release 4.1.1	SunOs 4.2 Release 4.1.1
Main Memory	40 MB	16 MB
Disk	WRENV IV 94181-385h with 320 MB	none
Disk controller	Emulex MD 21	none

Table 5: Description of the systems which performed the benchmark with GemStone

GemStone provides means for its configuration. Therefore, users can adjust a number of system parameters in order to configure GemStone's behaviour with respect to their needs. Table 6 describes those parameters that influence the performance and indicates how they have been adjusted for running the Merlin benchmark.

Parameter	Value	Description
`CONCURRENCY_MODE`	`FULL_CHECKS`	Used to control degree of concurrency allowed in terms of conflict detection. `FULL_CHECKS` assure, that transactions are performed in isolation.
`DBF_PRE_GROW`	`FALSE`	Decides whether physical disk space is allocated for GemStone databases once at all or on demand. `FALSE` indicates that space is allocated on demand.
`GEM_PAGE_CACHE_SIZE_KB`	3000	Sets the size of main memory used for caching objects within gem processes to about 3 MB.
`STN_PAGE_CACHE_SIZE_KB`	1000	Sets the size of main memory used for caching objects within the stone process to about 1 MB.

Table 6: Settings of GemStone Configuration Parameters

We performed the benchmark on the above given machines while they were running in multi-user mode. The multi-user mode was required by the RPCs needed for the remote mode. Besides the benchmark, there were no other loads on the machines. Moreover, the machines were the only machines in the network that had any load.

5 Results obtained with GemStone and GRAS

This section indicates the main results we obtained from performing the Merlin benchmark on top of GemStone and GRAS. Therefore, we discuss in the first subsection databases' efficiency with respect to physical disk space used. The main focus of this section however is on discussion of execution time required by the OMSs in different configurations in the second subsection. In the last subsection we discuss the impact of our results on construction of syntax-directed tools.

5.1 Space Efficiency

We performed the benchmark on both systems with several sizes of the initial database. They vary with respect to the number of levels as defined in subsection 2.4. Table 7 indicates how many modules have been created in order to set up initial databases with different levels.

Levels	Number of Modules
3	7
4	15
5	31
6	63
7	127
8	255

Table 7: Number of Modules in different initial Database States

Figure 13 depicts a comparison of the physical disk space used by GemStone and GRAS with the space that were occupied if the modules would be stored as flat text files in the Unix File System. Bars in that figure represent the difference between the physical space occupied by the database/file system with and without an initial database. The figures have been measured using the Unix command `du` on the files/directories where the resp. files/databases are stored.

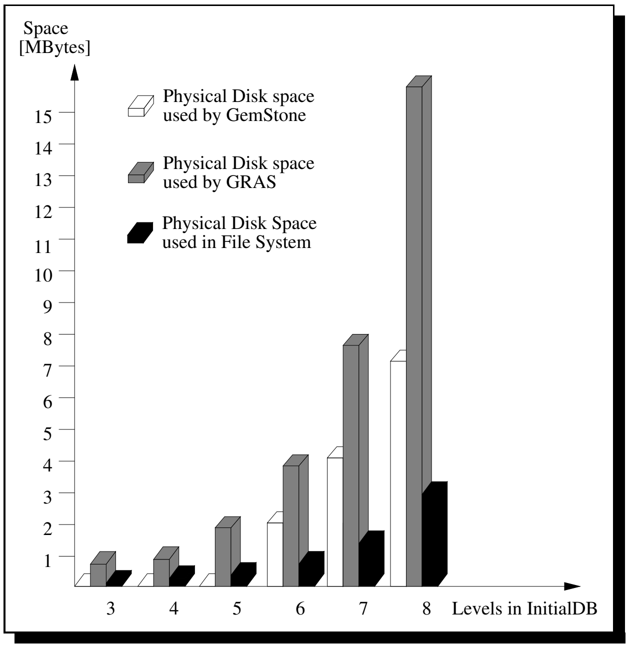


Figure 13: Disk Space used by GemStone and GRAS in Comparison to File System

In the configuration (c.f. table 6) in which we performed the benchmark, GemStone physically allocates disk space on demand in chunks of 1 MByte. In opposite to that 2 KBytes are the smallest unit to allocate disk space in the Sun OS file system.

As we can see from figure 13, GemStone need not allocate any disk space during creation of the initial database in levels 3–5. Thus, initial databases of that sizes stored in GemStone must considered to be smaller than a MByte. For the databases containing more than 5 levels, it used roughly 2.5 times as much disk space as the unparsed modules stored in the file system. GRAS used about twice as much disk space than GemStone and even five times the space needed for storing an unparsed module in the file system.

The mean value of physical disk space used for a module of the initial database at level 8 in the file system would be 11.5 KBytes. This value for GemStone is about 28 KBytes and in GRAS it is about 62 KBytes.

5.2 Time Efficiency

In this subsection we present an excerpt of the results we gained from executing the benchmark in a number of different configurations.

First we compare the performance of increment operations on top of GemStone and GRAS accessing a large initial database. In addition to these increment operations which must be executed very quickly, we compare the performance of analysis operations in GemStone and GRAS. After that we investigate the dependency of operations' execution times from the size of the initial database in GemStone and GRAS. Then we discuss the times needed for committing transactions in GemStone. Finally, we indicate how fast increment operations perform on top of GemStone in local mode compared to remote mode.

5.2.1 Increment Operations in a large Database

Figure 14 indicates the execution times needed for the increment operations with an initial database containing eight levels of modules. The number of modules created during a benchmark operation in this configuration was set to 10. The figures are the mean values from the results of 15 repeated execution of the benchmark. They show the elapsed real time of the benchmark operation in milliseconds divided by the number of increments that have been expanded, changed or deleted.
We compare the behaviour of GRAS with that of GemStone in local mode as GRAS also operates on a local disk. Furthermore, the times needed for commit of a GemStone transaction is for comparison purposes not included here, as modifications in GRAS are only persistent, after all graphs have been closed. Instead, the commit times needed by GemStone transactions are subject of subsection 5.2.4.

GRAS needs for a number of these increment operations more than 500 milliseconds which is untolerably high. All these operations have in common, that they have to access nodes and edges in other graphs. This is needed for checking whether the operation violates consistency constraints or in order to propagate changes to other graphs. Therefore, those graphs must be opened, accessed and closed afterwards. Note, that they can not be kept open all time due to earlier mentioned limitations of GRAS. In order to implement **CrImpObj**, for instance, we have to open the graph from which we intend to import and lookup whether the respective object is exported there. Then we have to enter the importing module into the graph's uselist and finally we must close the graph. Most of the time required by these operations is needed for opening and closing graphs.

None of the increment operations on top of GemStone performs slower than 75 milliseconds, which is sufficient for constructing syntax-directed tools. Note, that each operation also performs at least as fast as the equivalent operation on top of GRAS.
The operations which perform slowest are those that access the global dictionary `ArchSymTab`. That dictionary contains in this configuration about 5,000 entries. It is accessed in some benchmark operations (**CrModul**, **CrModTyp**, **CrExpOpe**, **ChModNam**, **ChModTyp**, **ChOpe-**

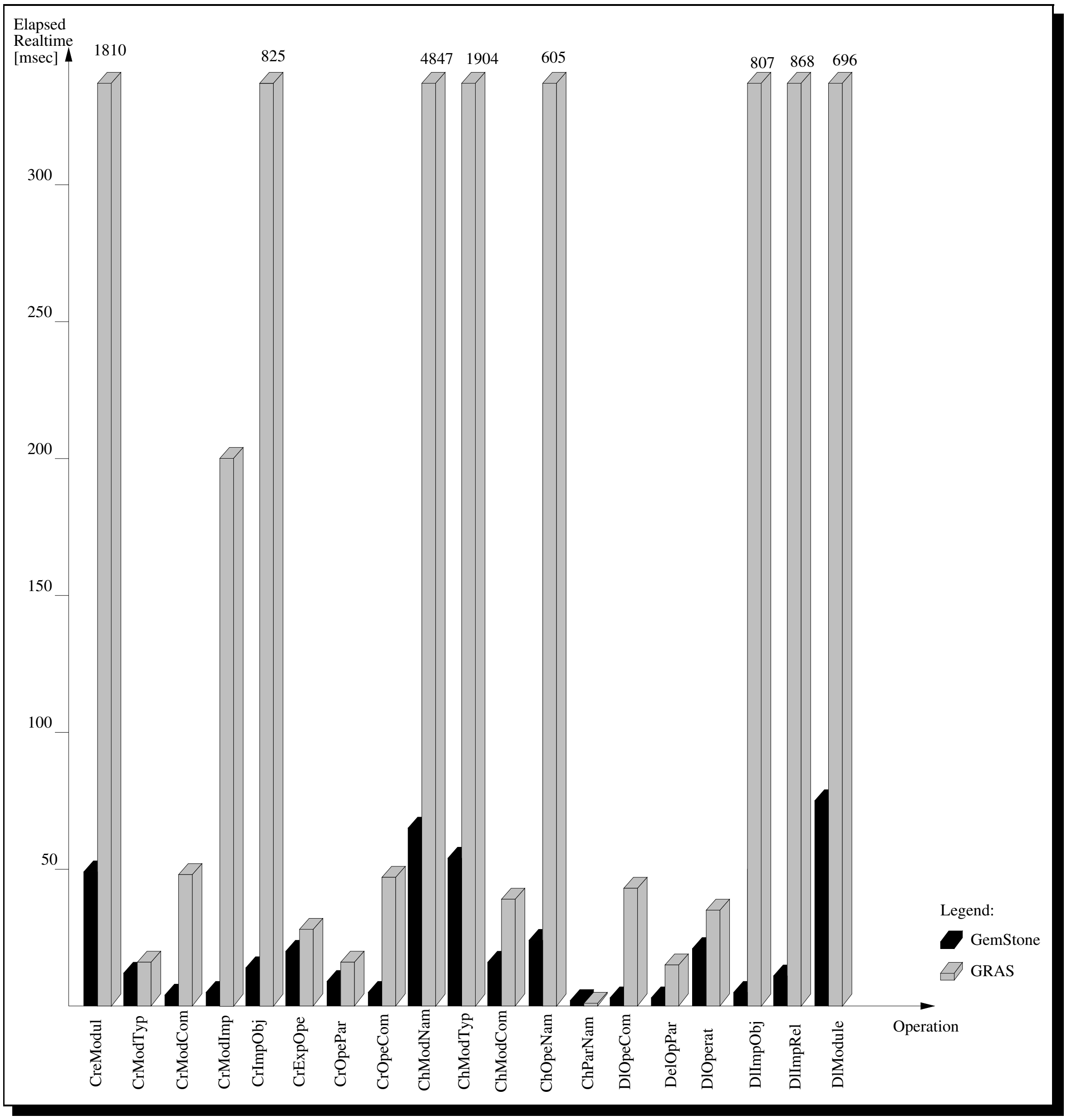


Figure 14: Increment Operations with an eight-level Database

Nam, **DlOperat** and **DlModule**) in order to decide about uniqueness of identifiers, to enter new identifiers, or to delete identifiers that do no longer exist. In create operations, the dictionary is accessed twice (querying for uniqueness and entering the new identifier), in change operations the dictionary is accessed three times (querying for uniqueness, removing the old identifier and entering the new one) and in delete operations the dictionary is accessed once (remove the identifier of the deleted object).
Execution times of operations on comments and parameters, which do not participate in complicated consistency constraints, are that low that they can be neglected.

5.2.2 Analysis Operations in a large Database

Figure 15 depicts the performance of analysis operations in an eight-level database. These operations are presented separately from increment operations, as they will not be used that often and thus their performance is not that critical. However, they allow us to draw conclusion about the performance we can expect from navigation-intensive operations.
The results shown here have been obtained from exactly the same configuration as described in the previous subsection.

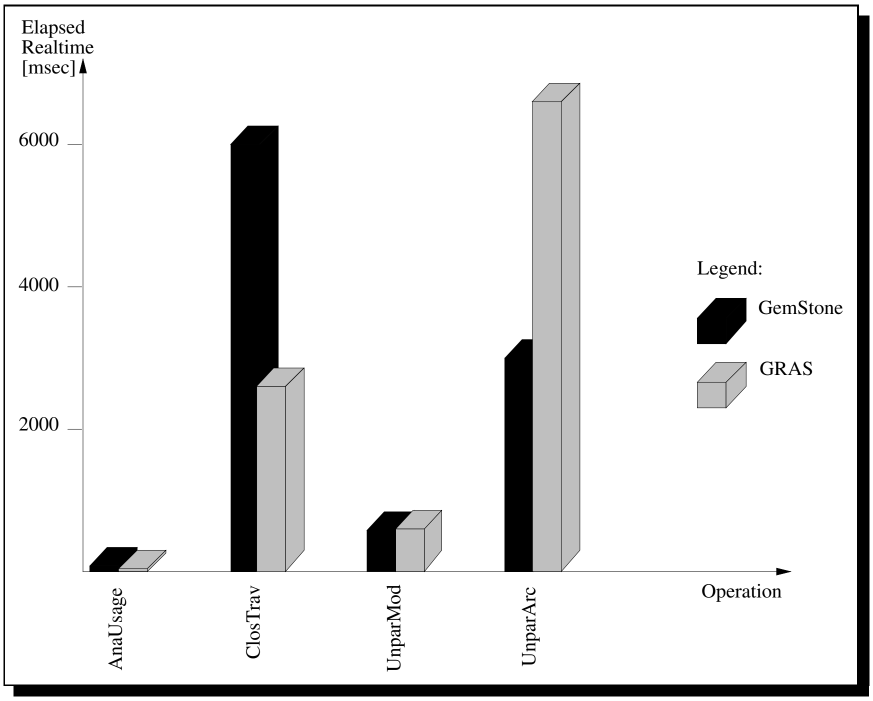


Figure 15: Analysis Operations with an eight-level Database

The performance of **AnaUsage** in GemStone and GRAS can be considered as uncritical. GemStone needs 14 milliseconds to compute the set of those modules which import from a module of the bottom-most level, whereas GRAS answers this query in 11 milliseconds.

ClosTrav is a kind of recursive query, that computes the transitive closure of the usage relationship between modules. In spite of the structure, we have chosen for the database, i.e. the root module transitively importing from nearly any other module, and in view of its size, a performance of less than 6 seconds seems to be tolerable.
However, it is worthwhile to note that GRAS performs this operation about twice as fast as GemStone. The implementation of this operation using GRAS benefits from the redundant storage of import information in the system graph. It can retrieve the set of importing modules (which is the major recursion step) by a single partial match query. In opposite to that, the GemStone implementation requires a number of navigation operations through a module's syntax graph in order to retrieve this set.

UnparMod performs with both OMSs in nearly the same time (599 milliseconds in GRAS and 579 milliseconds in GemStone). The execution time of this operation is of particular interest for building syntax-directed editors on top of OMSs. It visits all nodes of a module's syntax-graph, and retrieves all attribute values of terminal nodes like identifiers or comments.

Thus, it indicates the time required for initialisation of an syntax-directed editor's temporary data-structures during editor session start-up. These data-structures are needed basically, for displaying the unparsed representation of a module to the user and for identifying selected increments in terms of nodes of the syntax-graph.

The impact of the **UnparArc** operation is the same for an graphical architecture editor as that of **UnparMod** for a syntax-directed module interface editor.
UnparArc requires with GRAS more than twice the time as with GemStone (6.8 seconds vs. 3 seconds). The only explanation we have for this, is that GRAS' set operations which are used extensively for implementation of this operation tend to become slow with large sets being involved. This was the case in the configuration we used to retrieve these figures, as the set containing the modules in the architecture includes 265 modules.
Not only in GRAS, but also in GemStone the performance of **UnparArc** must be considered as too slow. Fortunately it is quite seldom, that we have to manage graphical documents of this size (265 modules with about 550 import relationships in a graphical document are also confusing from the user's point of view.). We would rather structure such documents into a hierarchy of sub-documents, which in turn are easier manageable for both, the user and the OMS. We come back to **UnparArc**'s performance in configurations with fewer modules in the next subsection.

5.2.3 Dependency of Execution Time from initial Database Size

We have performed the benchmark on both OMSs with initial databases of different sizes. In particular, we have varied the number of levels in the database between three and eight.
We found that the execution times of benchmark operations varied merely in the range of measurement faults only. There were, however, two exceptions that we discuss now: Firstly, **OpenOMS** and **CloseOMS** in GRAS and secondly, those analysis operations that access different number of objects depending of the initial database size.

OpenOMS in the GRAS implementation basically consists of opening the system-graph. Similarly, **CloseOMS** only closes the system-graph. The execution time for the former operation varies between 630 milliseconds in a three-level database and 3,200 milliseconds in an eight-level database. The performance of closing the system-graph is even worse: it varies between 1,020 milliseconds in a three-level database and 34,270 milliseconds in an eight-level database. The system-graph needs to be closed whenever a save-operation is performed. From a user's point of view, it is unacceptable to wait for a save-operation even of a large database for more than half a minute! One major motivation for the use of an underlying central OMS, is to overcome these response-times that are usually typical for load- and store-approaches.

Figure 16 depicts the dependency of the execution of **UnparArc**, **ClosTrav** and **AnaUsage** of the initial database size. It shows that **UnparArc** and **ClosTrav** in both OMSs become slower as the initial database grows. In opposite to that in both OMSs, execution time of **AnaUsage** slightly decreases asymptotically to 10 milliseconds.
The reason for the latter observation is, that the mean value of modules using a module of the bottom-most level decreases, if the number of modules at the bottom-most level increases.

UnparArc and **ClosTrav** become slower as the number of objects they access increase linear with the number of modules in the initial database.
The response times of these operations become critical from a user's point of view when they reach 1,000 milliseconds. GemStone performs both operations in less than a second in

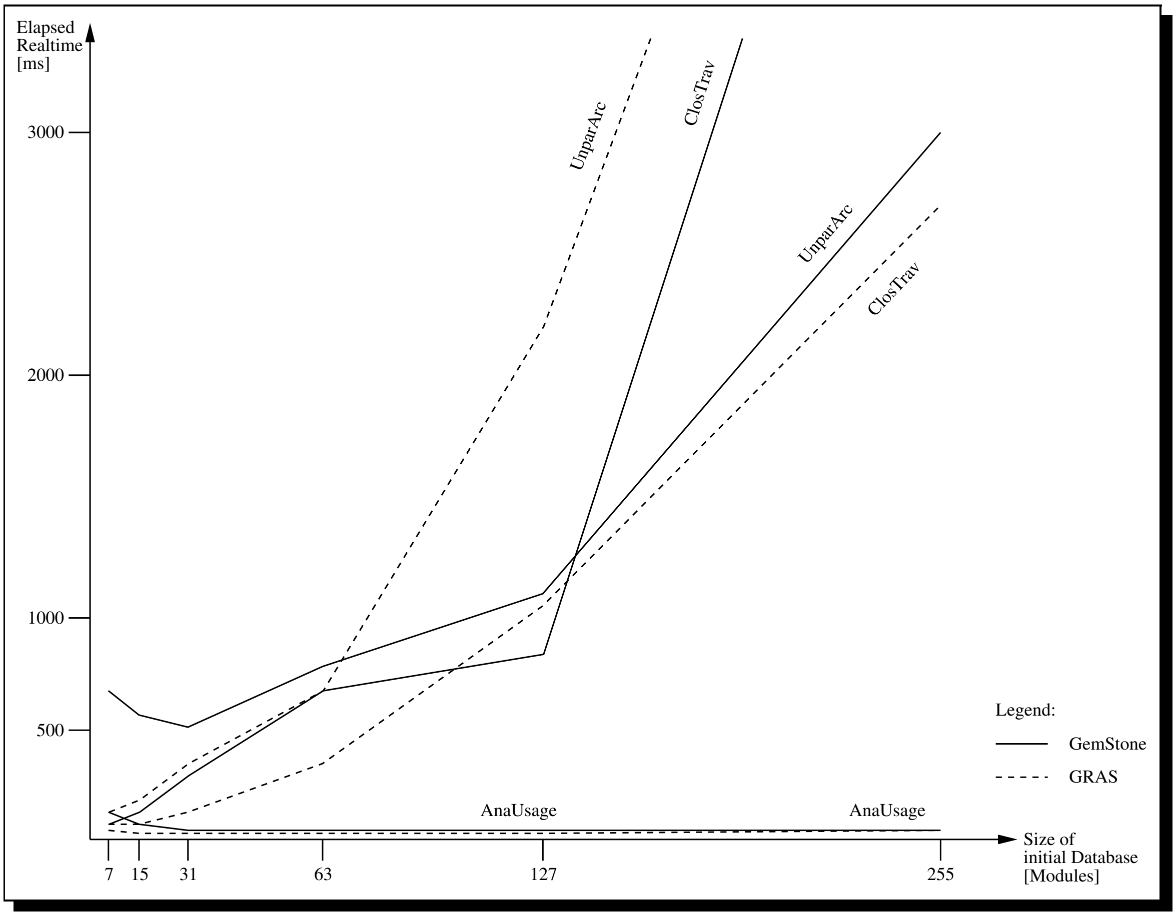


Figure 16: Impact of Database Size to Performance of Analysis Operations

databases with up to seven levels, i.e with 127 modules. GRAS performs **UnparArc** in a second only in databases with up to six levels.

5.2.4 Committing GemStone's optimistic Transactions

The aim of figure 17 is to depict what would happen to the response time of incremental user-interactions, if we would execute them as single database transactions.

We performed each benchmark operation accessing only a single increment as an optimistic transaction in GemStone. The figures depicted there have been obtained by performing each operation in this way 15 times with an eight-level database. The black part of a bar represents the time needed for execution of the operation itself and the grey part represents the time needed for committing.

In all operations, the sum of both times is dominated by the time needed for committing. The amount of time needed for a commit is in this configuration between 600 and 800 milliseconds. It varies in a three-level database between 480 and 650 milliseconds. It does not increase linear if more objects are accessed. In a configuration where we, for instance, expanded 3,200 parameter names and as many parameter types within one transaction a commit took about 3000 milliseconds. However, the results we obtained, seem to be the upper bound of what would be acceptable for a user-interaction.

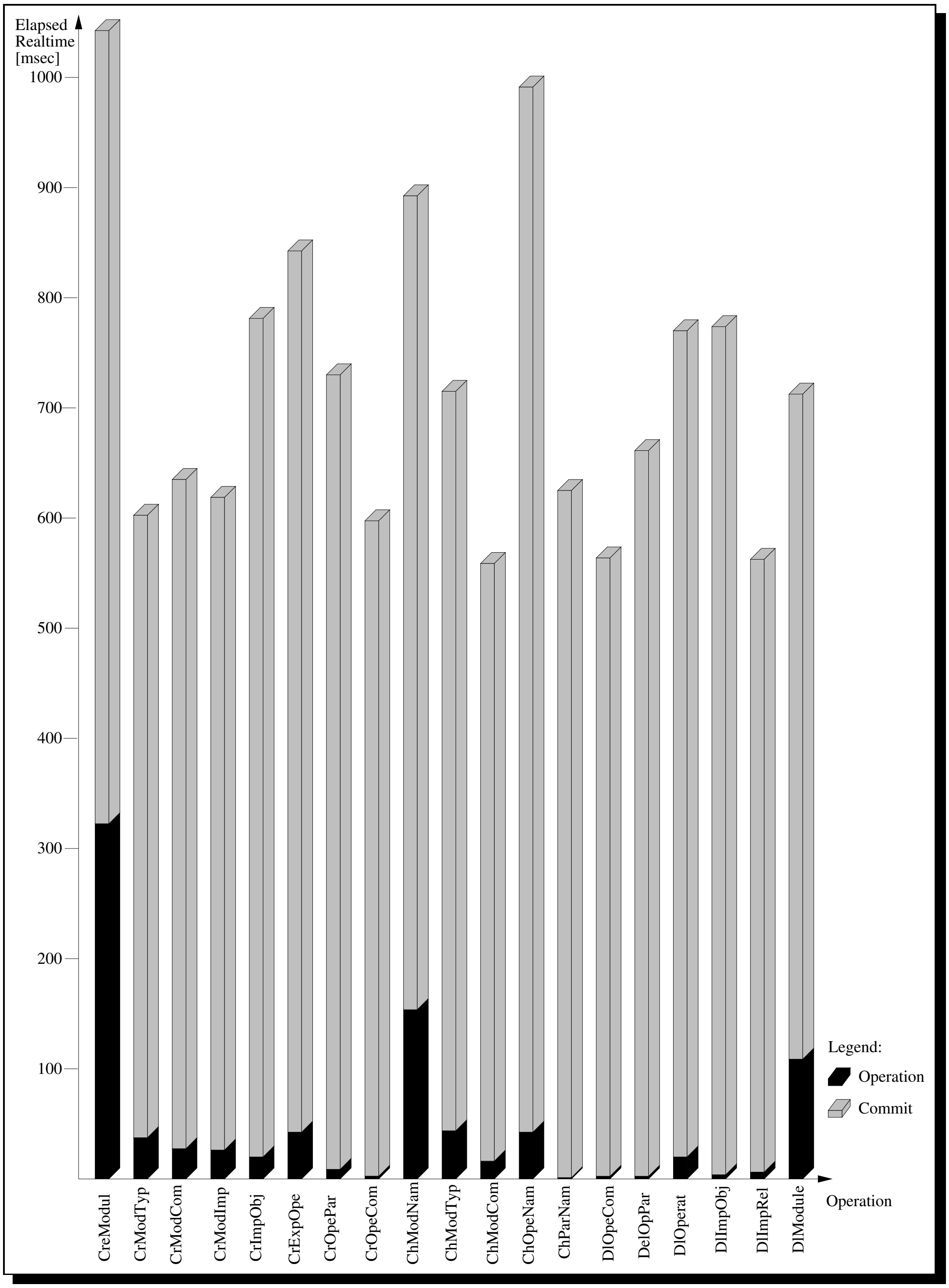


Figure 17: Performance of Commits in GemStone with optimistic Transactions

There is another interesting point in this figure to which we would like to draw reader's attention to. In **CreModul** and **ChModNam** we obtained a reasonable amount of time needed also for execution of the operation itself. The reason for that is the state of GemStone's caches. **CreModul** is defined in section 2.5 to be executed as the very first operation after database login. At that time there are of course no objects in GemStone's caches. Thus, execution of

CreModule causes a lot of physical disk accesses, some of which access very big objects. These accesses are expensive with respect to execution time. At the time execution of **ChModNam** starts, the cache is not empty but contains the wrong objects. This is due to the previously executed analysis operations. Especially **ClosTrav** and **UnparArc** massively access only architectural information like module names, module positions or import relationships which occupy the cache. Thus it is likely, that accesses to the global symboltable then performed by **ChModNam** cause a number of physical disk accesses.

5.2.5 A Comparison of GemStone in local vs. remote Mode

Figure 18 depicts a comparison of execution times of increment operations with GemStone in local and remote mode.

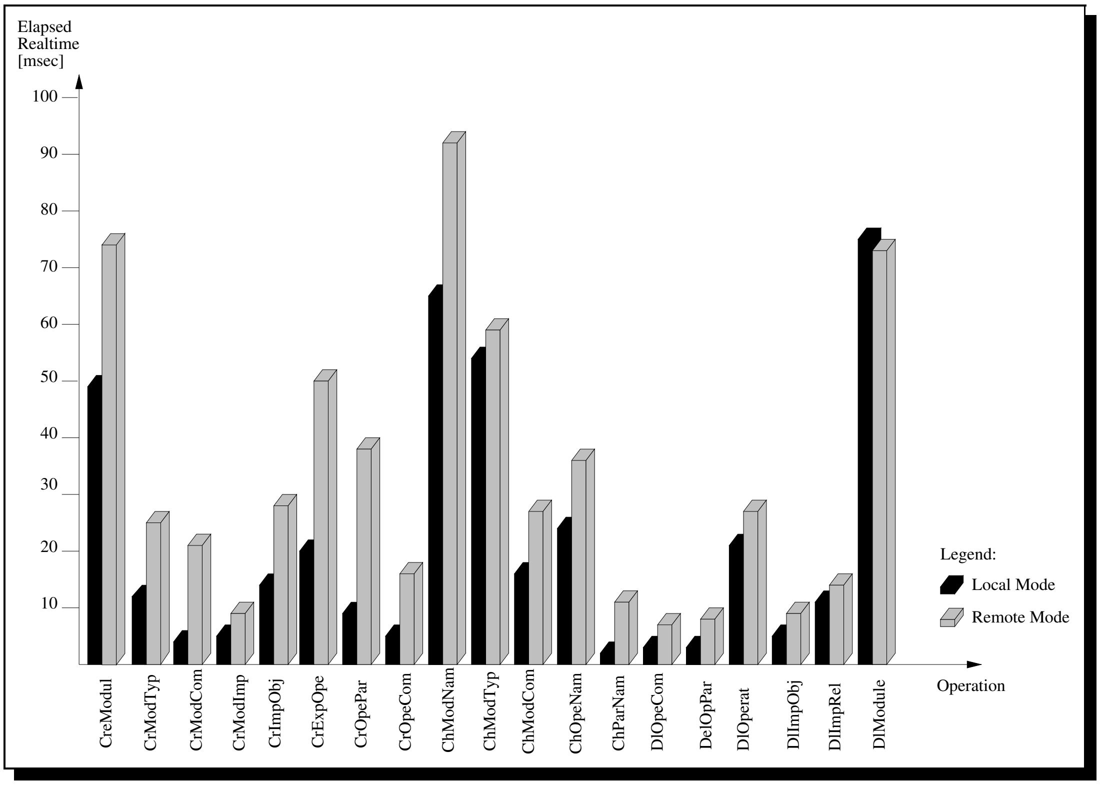


Figure 18: Increment Operations with GemStone in Local and Remote Mode

The figures have been obtained with an eight-level database. We repeated each operation 15 times and created ten new modules during the benchmark.

Generally, GemStone performs faster in local mode than in remote mode. The upper bound, however, for executing an increment operation in remote mode is 90 milliseconds. Moreover, we can see from that figure that execution in remote mode costs an overhead of at least five milliseconds which is about the time required for executing an RPC.

What is not depicted in that figure but worthwhile to mention are the execution times needed by **OpenOMS** and **CloseOMS**. To log into GemStone, which is the main operation performed by **OpenOMS** needs in local mode about 1.5 seconds whereas a login in remote mode costs about 11 seconds. This is due to the fact, that GemStone has to startup a Gem process

in remote mode, which it need not in local mode as the benchmark and the Gem process are linked together. Besides process startup, an additional amount of time is required for establishing the communication links between the benchmark and the Gem process.
Logging out of GemStone costs in local as well as in remote mode about 750 milliseconds.

5.3 Conclusions

OMSs are in general only comparable with respect to performance, if they provide the same functionality. Usually, the more functionality a system offers, the worse its performance becomes. GemStone has schema definition facilities, which do not only allow for a definition of the structure of objects, but also of objects' behaviour, it provides optimistic and pessimistic transaction management strategies and distributed access to objects by offering a client/server model for all of which GRAS has no counterpart. As GemStone offers a lot more functionality, it would be not surprising if GRAS would perform better! In this case, the comparison would be worthless.

Fortunately, things turned out exactly the other way around! Although GemStone offers a much richer functionality, it performs more efficient than GRAS. This not only holds for the disk space needed to store abstract syntax graphs, but also for the execution times of benchmark operations, which of course are more important. Except one operation, where GRAS benefited from redundant storage, GemStone could stand the comparison against GRAS.

Moreover, it turned out that GemStone is capable to perform an optimistic transaction executing a single increment operation in less than a second. A second for an user-interaction must be considered as too slow in general. An upper bound from the author's point of view would be about 500 milliseconds. However, during the past, hardware platforms got faster by a factor of two each year. We have seen a similar development with OMSs in general and GemStone in particular. Moving from GemStone Version 1.5 to 2.0 brought an increase in performance up to a factor of 5 [DEH+91]. Upgrading from Version 2.0 to 2.5 again brought a reasonable increase in performance. Thus, we expect in the near future GemStone to perform fast enough to really execute each user-interaction as a single transaction.

Our final conclusion concerns the overhead needed for remote access to a syntax-graph. This overhead is less than 30 milliseconds in all operations except database login. In the case of database login which is performed only once during an editing session, we can tolerate a response time of 10 seconds. In all other cases, the overhead is that small, that it is neglectable.

6 Summary and Future Work

We defined a new benchmark dedicated to measure the performance of OMSs when they are used as repositories for syntax-directed tools. Therefore, we defined the conceptual schema of the benchmark, the operations and initial database states based on an analysis of an existing syntax-directed environment.

We implemented the benchmark on top of the fully object-oriented database system GemStone and the graph storage system GRAS. We described those implementations, i.e. the realisation of the conceptual schema and that of the benchmark operations.

We described the results we gained from those two implementations with respect to place- and time-efficiency. It turned out, that although GemStone provides a lot more functionality, it does not perform worse than GRAS. With a number of benchmark operations it even shows better performance.

Currently, we implement the benchmark in a multi-user version in order to see at which user-load the performance of GemStone becomes unacceptably slow. Next, we intend to implement the benchmark on the fully object-oriented database system 0_2 [BDK92] in the single-user version we presented in this report, as well as in the multi-user version. This will then allow us to compare the performance of a server-oriented object-oriented database (GemStone) with that of a client-oriented object-oriented database (O_2).

Moreover, we participate in the Esprit-III project GoodStep (General Object-Oriented Databases for SofTware Engineering Processes). This project will enhance the O_2 system to meet specific software engineering application requirements. In this project, we intend to develop a generator for syntax-directed tools that store their documents in O_2.

Acknowledgements

We are greatful to our colleagues Dr. D. Schmedding, Prof. W. Schäfer, Prof. J. Welsh, J. Cramer, W. Deiters, Dr. S. Dewal, G. Junkermann and B. Peuschel for their comments on earlier drafts of this paper. We enjoyed using the Sodes-GRAS interface that was built at STZ GmbH and therefore, we would like to thank the OPUS team for their patient support in answering our silly questions. Moreover we are indebted to F. Buddrus for the tremendous effort he spent on the implementation of the benchmark on top of GRAS. A. Schürr did a great job when proof-reading the GRAS implementation w.r.t. the way we used GRAS. Last, but not least, we thank J. Rygula for configuring the machines as we needed them.

References

[ABD+90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of the* 1^{st} *Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japan.* Elsevier Science Publishers B.V (North-Holland), 1990.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: the Story of* O_2. Morgan Kaufmann, 1992.

[BL85] Th. Brandes and C. Lewerentz. GRAS: A non-standard data base system within a software development environment. In *Proc. of the Workshop on Software Engineering Environments for Programming-in-the-Large*, 1985.

[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.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proc. of SIGMOD Conference*, pages 316–325, Boston, MA, 1984.

[DEH+91] S. Dißmann, W. Emmerich, B. Holtkamp, K. Lichtinghagen, and L. Schöpe. OMSs Comparative Study. Internal Report D2.4.3-rep-1.0-UDO-EL, ATMOSPHERE, 1991.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and M. Mélèse. Document structure and modularity in Mentor. *ACM SIGSOFT Software Engineering Notes*, 9(3):141–148, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[DHK+90] S. Dewal, H. Hormann, U. Kelter, D. Platz, M. Roschewski, and L. Schöpe. Evaluation of Object Management Systems. Technical Report 44, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1990.

[ELN+92] 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.

[ES92] W. Emmerich and W. Schäfer. Dedicated Object Management Benchmarks for Software Engineering Applications. Technical Report 63, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1992.

[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.

[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.

[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.

[Nag85] M. Nagl. An Incremental and Integrated Software Development Environment. *Computer Physics Communications*, 38:245–276, 1985.

[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.

[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.