# Databases for Software Engineering Environments

—

# The Goal has not yet been attained

Wolfgang Emmerich*, Wilhelm Schäfer* and Jim Welsh‡

Informatik 10
Universität Dortmund
Postfach 500 500
W-4600 Dortmund 50
Germany
emmerich@udo.informatik.uni-dortmund.de
wilhelm@udo.informatik.uni-dortmund.de

Dept. of Computer Science
‡ University of Queensland
Queensland 4072
Australia
jim@cs.uq.oz.au

**Abstract**

We argue that, despite a substantial number of proposed and existing new database systems, a suitable database system for software development environments and especially process-centred environments does not yet exist. We do so by first reviewing and refining the requirements for such systems. We then review a number of available and archetypical database systems and show that they do not meet these requirements.

## 1 Introduction

Software development environments (SDEs) include tools which support most of the software life-cycle phases, i.e. construction and analysis of the corresponding documents and document interdependencies. Sophisticated integrated environments enable the incremental, intertwined and syntax-directed development and maintenance of these documents such that errors are easily traced back through different documents and necessary changes are propagated across document boundaries to correct the errors (c.f., for example, [HN86, ELN+92, BCD+88]). Such environments should provide multi-user support[1], i.e. they should have flexible and adaptable mechanisms (often called "design transactions") to control access by a number of users to shared information. The construction of such environments and corresponding advanced design transaction mechanisms is an area of active research (c.f., for example, [BK91, PSW92, TSY+88]). We call this latter kind of environment a process-centred environment (PSDE), because the provided multi-user support is or rather should be based on a well-defined development process, i.e. the definition of the users' responsibilities, their corresponding access rights, and the schedule of the activities to be carried out by them.

[1]As the people normally called software developers are the users of an SDE and this paper discusses only SDEs, we use the term user instead of developer hereafter.

For any kind of environment, a large number of objects and corresponding relations on very different levels of granularity have to be stored and retrieved, and, in case of a process-centred environment, these objects must be manipulated under the control of an advanced transaction mechanism. An underlying database management system is thus a key component of a PSDE and if not chosen carefully, could become a major performance bottleneck when the PSDE is in use.

As early as 1987, Bernstein has argued that dedicated database systems for software engineering, specialised with respect to functionality and implementation, are necessary [Ber87]. He, and others [TSY+88] argued that the functionality and efficiency of existing systems (in particular, relational systems) do not adequately support the construction of software engineering tools and environments. A number of systems, some of which differ radically from standard relational technology, have since been described in the literature and some are now available as commercial products.

In this paper we argue that, despite the substantial number of these new database systems, a suitable database system for SDEs, and especially PSDEs, does not yet exist. We do so by reviewing and refining, in sections 2 and 3, some of Bernstein's requirements based on our own experiences in building such environments and tools. Section 4 then briefly reviews a number of available database systems, and shows that these requirements are not met by them.

# 2 Process centred Software Development Environments

Architecturally, a process centred software development environment consists of the following main components:

- a process engine that coordinates the work of developers involved in a project,
- a set of integrated, syntax-directed tools that allow developers to conveniently manipulate and analyse documents and to maintain consistency between related documents of different types, and
- an underlying database for software engineering (DBSE) which is capable of storing project information and documents.

## 2.1 The Process Engine

The process engine executes a formal description of a software process in order to coordinate the work of the users involved in a project. In more detail this covers the following issues.

**Multi-User support** The process engine determines for each user participating in a project a personal agenda that indicates on which documents he or she may perform particular actions. The contents of the agenda depend on

- the user's responsibilities in the project and
- the current state of the project.

The invocation of tools that enable a user to perform the actions contained in an agenda is controlled by the process engine via the agenda. In simple terms, the agenda acts as a menu from which the user selects his or her next activities.

In most projects a number of users work in parallel on different parts of the overall project activity. This means that changes to a user's agenda are not only necessary due to his or her own actions. They are also necessary when some other user changes the state of a document on which the first user's agenda depends.
As each user usually works at a personal workstation, the agendas must be presented and updated in a distributed fashion. Likewise, the tools that are called via the agenda have to access documents in a distributed fashion.
To ensure orderly use of documents in parallel development activities, the process engine must also be able to create versions of documents, or sets of documents, to retrieve a particular version and to merge version branches of documents.

**Efficiency** The state of the project changes, when a new document is introduced or an existing document is deleted, when a document is declared to depend on some other document, when a document becomes complete or when it becomes incomplete due to a change in some other document it depends on. Although this list is incomplete, it already indicates that changes to project states occur frequently. All of them cause a recomputation of the agenda. As a user cannot perform any tasks while his or her agenda is being computed, the computation must be done efficiently.

**Persistence and Integrity** The process engine may need to be stopped from time to time. Therefore it must be able to store the state of the project persistently in order to prepare a restart. Even if it is stopped accidentally e.g. by a hardware or software failure, it must resume with a consistent project state. Moreover, such a failure must not result in a significant loss of project information. Thus we require that the process engine preserves integrity of any project information, i.e. it ensures that continuation of any operation is possible after any failure.

**Change** It has been widely recognised that software processes can not completely be defined in advance [Mad92, PS92]. The process being executed needs to be changed "on the fly" from time to time. For example, new users may participate in the project, responsibilities may be redefined, new types of documents may be introduced or new tools for manipulation of these documents may need to be integrated.

**Reasoning capabilities** As it might not always be clear to a user why a particular document is to undergo a particular action, the process engine should have the capability to explain this to the user. For instance, the process engine should be able to answer questions from a user like "What is the state of the specification of module m1?", "Who else is involved in the project?" or "What happens if I now code module m1?". The latter example indicates that the reasoning capabilities do not only explain how a particular project state was reached, i.e. the past (as in classical expert system reasoning) but that they also give insights about possible future consequences of a particular action.

### 2.2 Highly integrated syntax-directed tools

**Syntax-direction** The tools contained in a PSDE are used by users to edit, analyse and transform documents.
To give as much support to users as possible, the tools should be directed towards the syntax of the languages in which documents are written. In particular, they should reduce the rate at which errors concerning the context-free syntax are introduced to documents. If the editors are template-based, such errors can be detected by scanning only the terminal symbols input by the user. More sophisticated editors that enable textual input must parse the text during or immediately after each edit operation and indicate the errors if any. Such editors normally reject syntactic errors within newly inserted material, but may still permit syntactic inconsistencies to result from edit operations. Changing a Pascal if-statement into a while-statement by two separate word replacements is the classic example of a situation which demands such tolerance.

**Consistency preservation** Besides dealing with errors concerning the context-free syntax, tools should also deal with errors concerning both the internal static semantics of documents and inter-document consistency constraints. The tools may allow temporary inconsistencies to be created both during input and as the result of edit operations, since document creation in a way which avoids such temporary inconsistencies is impractical in many cases.
Once created, of course, these inconsistencies must be brought to the user's attention in an appropriate manner. Two approaches may be taken: automatic consistency checking, i.e. whenever an inconsistency is created it is automatically brought to user's attention, and consistency checking on demand, i.e. inconsistencies are checked for only when the user requires that it be done.
Tools should also support users in removing inconsistencies. In particular, follow-on inconsistencies such as use of a non-existing import, could be removed on demand by change propagation, such as propagating the change that redefines the import to all places where it is used.
Finally, tools must allow the user to define additional semantic relationships during the course of a project. For instance, dependencies between the source-code, the test plans and the technical documentation on the level of identifier names (and corresponding section titles in the technical documentation) may only be defined in this way.

**Persistence and Integrity** Obviously, documents must be stored persistently since they must survive editing processes. Moreover, users require tools to operate as safely as possible, i.e. in case of a hardware or software failure they expect that the integrity of documents (their immediate usability by the same or other tools) will be preserved and that significant user effort will not be lost. Thus, we require the persistence to be achieved as follows: An interactive session with a tool of the PSDE consists of a sequence of user-actions some of which may change the software document under manipulation. We require from tools that a user-action is persistent if and only if it is completed. Moreover, user-actions must be designed in a way that the integrity of a document is guaranteed whenever a user-action is completed. In case of a hardware or software failure we require that the tools should recover to the last completed user-action.

**Backtracking** When the consequences of user actions are persistent, users must have the ability to backtrack or "undo" such actions when mistakes are made. Thus, in addition to the document versions used for management purposes in a multi-user project, users may want to store intermediate revisions of a document so that they can revert to a previous revision if their subsequent modifications turn out to be ill-chosen. In general, tools must support this user ability to backtrack, either directly (via explicit undo mechanisms) or indirectly (by enabling users to make their own provision for backtracking, as in the use of revisions outlined above).

**Efficiency** Very fast typists can type about 300 characters per minute. In this case, the time between successive keystrokes is about 200 milliseconds. Thus any response time of an editor below 200 milliseconds is non-critical, as users are never going to recognise them as delays.
Unlike secretaries, users do not type continuously. They frequently pause to think about what to do next. These thinking periods break the basic sequence of user transactions into a higher-level sequence of task-oriented transactions. If the non-trivial processing that a tool carries out is aligned with these natural breaks in user interaction, much higher response times may be acceptable. Thus editor users frequently pause when they have finished editing one syntactical construct before they start editing another. If the processing involved in static semantic checking, say, is somehow aligned with these syntactical constructs, users will rarely recognize a response time for this processing that is below a second as a noticeable delay.
In other circumstances, users may accept much higher response times, if they occur less frequently and can be justified by the complexity of the task concerned [WBK91]. No user, for instance, would reject using a compiler just because it needs more than a second to compile a source. In general, therefore, the response times achieved by tools for various operations must be consistent with the user's needs and expectations at keystroke, transaction and task levels.

# 3 Requirements for DBSEs

## 3.1 Persistent Document Representation

**What is a document in the database?** The common internal representation for syntax-directed tools such as syntax-directed editors, analysers, pretty-printers and compilers is a syntax-tree of some form. In practice, this abstract syntax-tree representation of documents is frequently generalised to an abstract syntax-graph representation for reasons such as efficient execution of documents, consistency preservation by tools, and user-defined relations within documents.

As an example of a small excerpt from an internal Modula-2 program representation c.f. figure 1 which will be used to explain further details in the following.

Each PSDE supports execution of documents to some degree. In some PSDEs, the process model is the only executable document, but most PSDEs offer also test and execution of software documents by interpretation of the document syntax-tree. Such interpretation can be achieved purely in terms of an attributed syntax-tree itself but is very inefficient. For executable documents, therefore, the abstract syntax-tree representing the document is commonly enhanced by additional edges that indicate the control flow in order to enable more efficient interpretation (c.f. the "- - ►" edges exemplifying the control flow of a while statement in figure 1).
Consistency checking on a document can be done by attribute evaluations along parent/child paths in the document's attributed abstract syntax tree. The evaluation paths are computed

```
PROCEDURE InitWindowManager(y:<TypeIdentifier>);
VAR x:<TypeIdentifier>;
BEGIN
  WHILE <BooleanExpression> DO
    x:=Expression;
    <Assignment>
  END
END InitWindowManager;
```

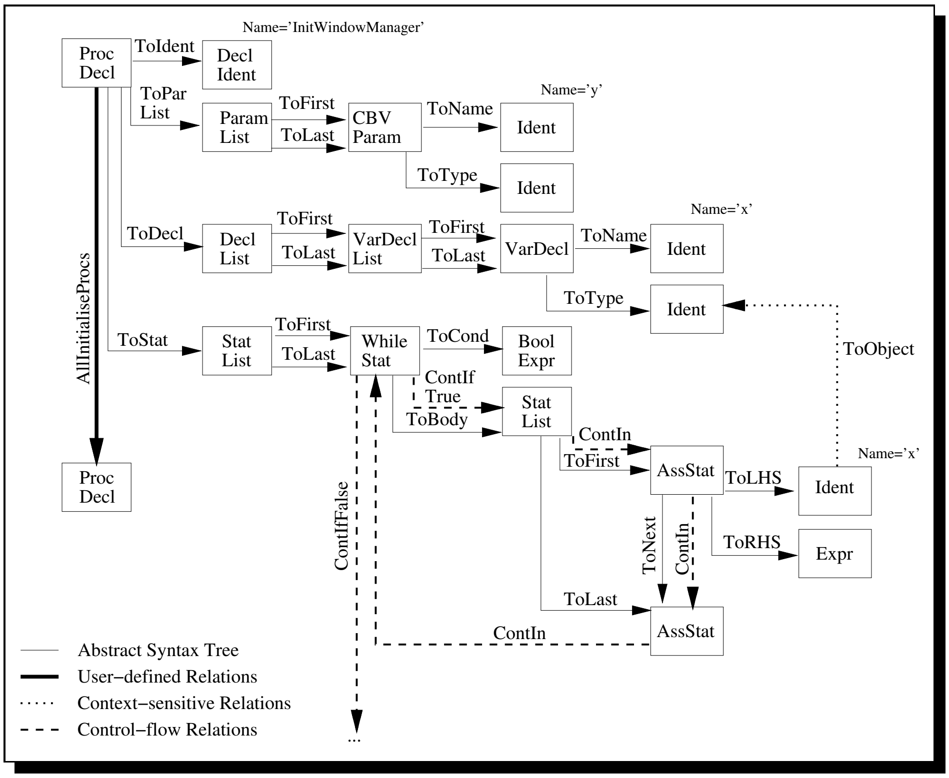


Figure 1: Modula-2 external textual and internal graph representation

at generation-time based on attribute dependencies. This approach, however, has proven to be quite inefficient even for static semantic checks of a single document, because of the long path-lengths involved. Techniques based on the introduction of additional, non-syntactic paths for more direct attribute propagation have been developed [JF82, BC85]. Such non-syntactic paths are examples of context-sensitive relationships which connect syntactically disjoint parts of a document, and are used in both consistency checking and change propagation when the document is changed (c.f. the "·····►" edge which connects declaration and use of an identifier in figure 1).

As noted in section 2.2, the user may also introduce additional user-defined relations between document parts for purposes of documentation, traceability, etc.. All such relationships must also be seen as edges in the abstract syntax graph that represents the document during manipulation, as illustrated by the "──►" edge labelled *AllInitialiseProcs* which connects all procedures which implement create/initialise operations.

**How is it stored?** Due to the requirements of persistence and integrity, a persistent representation of each document under manipulation must be updated as each user-action is finished. Typically a user-action affects only a very small portion of the document concerned, if any. Given that the representation under manipulation is an abstract syntax-graph, however, the update can easily become inefficient if, firstly, a complex-transformation between the graph and its persistent representation is required and, secondly, the persistent representation is such that large parts of it have to be rewritten each time, although not being modified. This would for instance be the case, if we had chosen to store the graph in a sequential operating system file which is updated at the end of each user-action.
Such inefficiency can be avoided completely if the persistent representation takes the form of an abstract syntax graph itself, with components and update operations that are one-to-one with those required by the tools concerned. To allow this approach, therefore, the DBSE must support the definition, access and incremental update of a graph structure of nodes and edges with associated labelling information. To preserve the integrity of the abstract syntax-graph, the DBSE must support atomic transactions, i.e., a transaction mechanism that allows us to group a sequence of update-operations such that they are either performed completely or not performed at all. To ensure that a tool can recover in case of a failure to the state of the last completed user-action, each completed DBSE transaction must be durable.

**Inter-document relationships** Context-sensitive relations and user-defined relations between document components, as discussed above, contribute to the need for an abstract syntax-graph representation of a software document. In practice, however, such relations are not confined to within individual documents – they frequently exist between components of distinct documents. As an example c.f. figure 2.

The figure illustrates the connection of the previously shown source-code graph (figure 1) with the corresponding parts of a design document and the technical documentation resp. Thus a module definition in the design document has inter-document relationships with the implementation of that module in the corresponding implementation document. Likewise, a paragraph in the technical documentation may be linked to the corresponding (formal) module definition in the design document, for traceability reasons.

To handle these inter-document relationships in a consistent way, the obvious strategy is to view the set of documents making up a project as a single project-wide graph. This approach, however, immediately reinforces our concern that the cost of updating its persistent representation should be independent of the overall size of the graph concerned, and requires that the DBSE must avoid imposing limits on the overall size of the graphs it can handle.

We note that this generalisation to a single project-wide graph does not necessarily undermine the concept of a document as a distinguishable representation component. If we distinguish between *aggregation* edges in the graph which express syntactic relationships, and *reference* edges, which arise from control, context-sensitive or user-defined relationships, then a document of the project is a subgraph whose node-set is the closure of nodes reachable by aggregation edges from a document node (i.e., a node not itself reachable in this way), together with all edges internal to the set[2]. The edges not included in this way are then necessarily the inter-document relationships inherhent in the project.

[2]What we call a subgraph here, is comparable to the notion of a *composite entity* in PACT VMCS (cf. [Tho89])

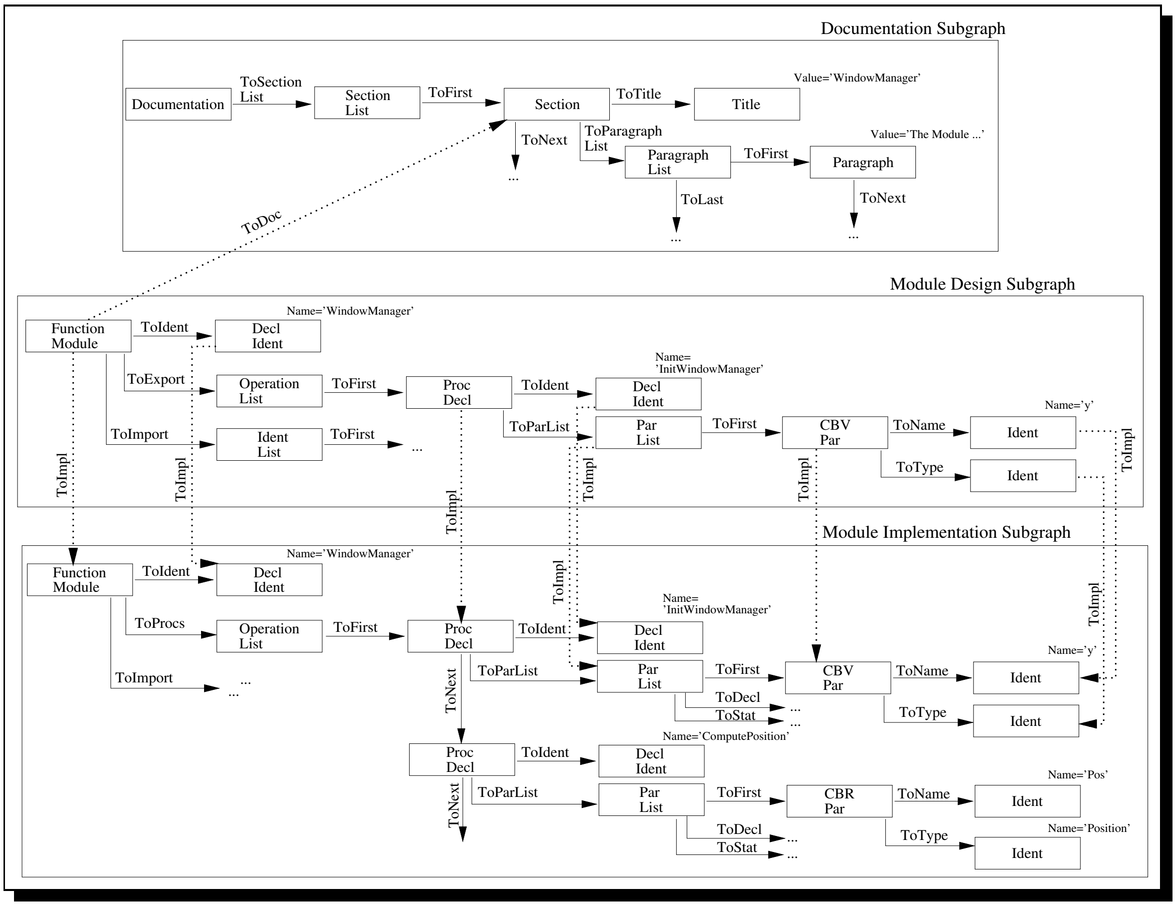


Figure 2: Internal representation of Inter- and Intra-document Relations

## 3.2 Data Definition Language/Data Manipulation Language (DDL/DML)

The kinds of nodes and edges required to represent a project, and the attribute information associated with each, cannot be determined by the DBSE itself. It should be defined and controlled, however, by the DBSE in order to have different tools sharing a well-defined project-graph. The overall structure of the project's syntax-graph should therefore be defined in terms of the data definition language of the DBSE and be established and controlled by the DBSE's conceptual schema.
As a minimum, we require that the data definition language can express the different node types that occur within the graph, that it can express which edge-types may start from node types and to which node types they may lead, and that it can express which attributes are attached to node types. Such basic requirements are common to any graph storage.
In practice, the data definition language should be tailored to the syntax graphs that the DBSE is used to store. Structures that occur often in syntax-graphs are lists and sets of nodes that contain nodes of possibly different types. As an example, consider statement lists in programming language documents that contain statements of different types or dataflow diagrams that contain sets of terminators, processes and stores. The data definition language should therefore offer means to express these common aggregations as conveniently as possible.

As argued previously, changes to the internal syntax-graph should become incrementally persistent. Therefore, edit operations performed by tools on documents have to be implemented in terms of operations modifying the internal syntax-graph. These operations should be established as part of the DBSE schema for several reasons:

**Encapsulation** The structure definition of the project-graph should be encapsulated with operations which preserve the graph's integrity. They then provide a well-defined interface for accessing and modifying the graph. In order to enforce usage of this interface, the operations must become part of the DBSE schema. Otherwise tools could circumvent the interface and directly access and modify the graph using low-level DBSE operations such as creation or deletion of nodes and edges.

**Decoupling of tools** As argued previously, relationships between documents should be maintained. If graph modifying operations were implemented in tools rather than as part of the DBSE schema, tools would have to know the document structure of documents manipulated by other tools in order to establish or change these inter-document relationships. This would unnecessarily lead to dissemination of information about the project-graph's structure into the implementation of tools.

**Performance** Executing graph accessing and modifying operations within the DBSE is more efficient than executing similar operations within tools as the number of nodes and edges that need to be transferred from the DBSE to tools via some network communication facility is reduced significantly.

To establish graph-modifying operations as part of the DBSE schema, the DML must be powerfull enough to express them. This means in particular, that the DBSE's DML must be capable of expressing creation and deletion of nodes and edges as well as assignment of attribute values. Moreover, the DML must be computationally complete, as alternatives and iterations are needed in graph-modifying operations for navigation purposes.

In addition, we noted in subsection 2.1 that the user may wish to query the project state via a reasoning component in the process engine. In practice, such queries may also arise from within the process engine itself, and from the process engineer who maintains the process governing any project.
The process engine needs to query the project's current syntax graph in order to extract information about the states of documents on which the engine must base its decisions. An example for such a query could be: select all program modules from the set of modules for which programmer $p_1$ is responsible that are incomplete but whose specifications are complete.
The process engineer may want to query the internal project graph in order to determine the project state.
The queries to be answered by the reasoning component are not known in advance, they have to be formulated in a process-related query language and must be translated by the reasoning component into a DBSE query. Likewise the queries the process engineer will use are not known a priori, i.e. at the time the PDSE is being built, so they cannot be precompiled. Thus, the DBSE must offer ad-hoc query facilities to be used by the process engine, and indirectly by the user and the process engineer.

### 3.3 Schema Updates

In section 2.1 we noted that the process being executed may have to be changed "on the fly". The DBSE must therefore enable the definition of new types of nodes and edges that are to be included in the internal syntax graph. Existing nodes and edges must not be affected by this kind of schema update.

Moreover, it is necessary that new types of edges can be added to existing types of nodes to allow the integration of nodes of newly defined types into an existing syntax graph. This implies that existing nodes whose types are modified by such a schema update can migrate from the old type to the modified one.

Consider as an example, the introduction of a new quality assurance procedure to be applied to all future documents as well as existing ones. The new procedure may require that the person who reviews a document writes a review report that is attached to the document. Furthermore his or her name is to be recorded as an attribute of the reviewed document. In this situation, the schema of the internal syntax graph needs to be modified: A new document type *review report* must be included and each reviewable document needs to have an additional node to record the document's reviewer and an additional edge to express its relationship to the review report.

### 3.4 Revisions and Versions

Given the overall representation of a project defined in section 3.1, the DBSE must support the creation and management of versions of those subgraphs that represent versionable document sets. In particular, it must enable its clients to derive a new version of a given subgraph, to maintain a version history for a subgraph, to remove a version of a given subgraph, and to select a current version from the version history. In doing so it must resolve between alternative duplication strategies, both within versions and for extra-version relations, preferably in a definable way.

Within versioned subgraphs the DBSE must resolve between fully lazy, fully eager and hybrid duplication strategies for the nodes and aggregation edges of the subgraph concerned. Fully lazy duplication gives maxium sharing of components, and hence minimum storage utilisation, but complicates the update process during user edits. Fully eager duplication avoids all such complications, but implies maximum storage utilisation. To meet the needs of all PSDE functions, a definable hybrid strategy between these two may have to be suported by the DBSE.

The DBSE must also resolve between alternative strategies for handling both intra- and inter-document relationships (or reference edges). Within a document, reference edges will normally be treated (like aggregation edges) as *version-duplicated* (i.e., a new edge is automatically created for each version created). In particular circumstances, however, such edges may be seen as *version-specific* (and hence not duplicated when new versions are created). Relations between documents within a versioned document set (i.e. within a configuration) are treated similarly, i.e., they may be either version-duplicated or version-specific. Relations between a versioned document and a document outside the versioned set are also subject to the same basic choice, but version-duplication in this leads to a state which we call *version-transparent* since the same relation necessarily holds between all versions within the versioned set and the same (component of the) unversioned document. Again a definable hybrid strategy may be necessary for effective PSDE support.

Figure 3 depicts the logical view of the module design graph being under version control. An additional (version-specific) edge *ToPredVers* is introduced that makes the version history explicit by leading for each document node to the document node of the predecessor version. An example of a version-transparent edge is the *ToDoc* edge connecting a function module node with the corresponding section title node in the documentation subgraph.

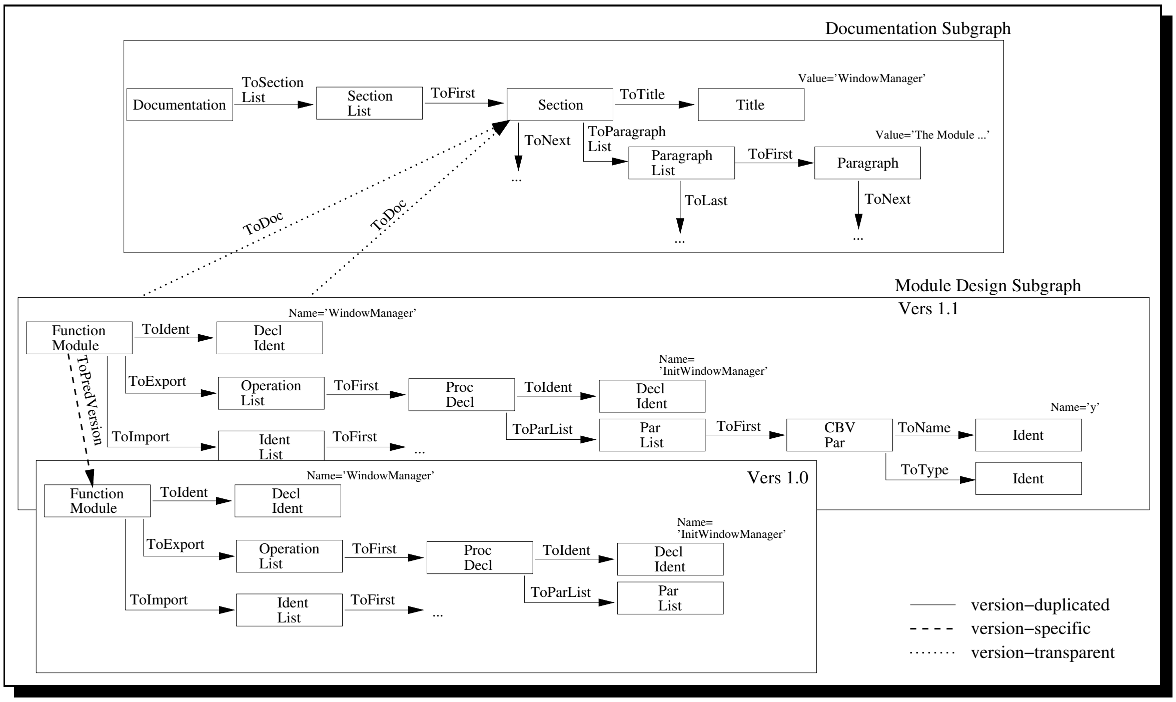


Figure 3: Logical View of a Document under Version Control

## 3.5 Access Rights and Adjustable Transaction Mechanisms

Requirements on the DBSE functionality which stem from the multi-user support of PSDEs, are the definition of access rights for particular documents and parts thereof and the definition of a variety of transaction mechanisms to control and enable parallel access to shared information by multiple users.

In more detail, to define access rights, the DBSE must be able to identify users and arbitrary many (probably nested) user groups. Secondly, the DBSE must allow to define and modify the ownership of subgraphs (which could even be a single node) at any time. Similarly, a subgraph may be accessed by several groups. Thus a subgraph needs to maintain its group memberships. Thirdly, the DBSE must allow to define and modify access rights for a subgraph individually for its owners and groups at any time. Finally, the DBSE must enforce that the defined access rights are respected by all its users[3].

Conventional transaction mechanisms which control parallel updates of the database have been proven to be too restrictive to be used in PSDEs because they could result in a rollback which deletes the effect of a possibly long-lasting human development effort, or they could block the execution of a certain activity for days or even weeks. Both situations are untolerable.

Consequently, advanced transaction mechanisms have been developed like group transactions [KSUW85], cooperating transactions [NSZ91], or split/join transactions [PKH89]. Their common characteristics is that they relax one or more properties of conventional transaction mechanisms which are atomicity, consisistency preservation, isolation and durability. For a detailed overview and critical evaluation of those mechanisms we refer to [BK91].

[3]The kind of discretionary access control required here is similar to that provided by modern operating systems like e.g. UNIX. It differs in that a subgraph can have more than one group that may access it. Thus, we can express access rights in a more flexible way without introducing artificial user groups.

We argue, however, that none of them is powerfull enough to be incorporated as **the** transaction mechanism into the database of a PSDE. As already argued in [PSW92], the process engine which knows the current state of an ongoing project, can only decide whether and when to request a lock for a particular subgraph and how to react, if the lock is not granted. It also defines whether a transaction is executed in isolation or in a non-serialisable mode.

The requirements for the transaction mechanisms of a DBSE are that it offers the possibility to define and invoke either a serialisable transaction or a non-serialisable transaction which only guaratees atomicity and durability.

Atomicity and durability are needed to preserve the integrity of the project graph against hard/ or software failures as required in subsection 2.1 and 2.2 resp. Serialisability is needed, for example, when project state information and corresponding user agendas are updated (c.f. subsection 2.1). This information must not be invalidated by parallel updates as that could hinder the process engine from continuing its work or let the user perform unnecessary work. Fortunately, these updates are relatively short and do not involve any human interaction. Moreover, computation of a users' agenda only incorporates read access to project's state information such that it may be performed in an optimistic mode.

An example of a situation where transactions are either executed in a serialisable or non-serialisable mode depending on the project state is the following. During the very first development of a document, editing the document could and should be done in isolation until the document has evolved into a certain mature state, maybe a state where the document is released. During maintenance phase, a released document should be always consistent in itself as well as w.r.t. other documents. Thus a transaction which does change propagations due to error corrections should be executed immediately, even if affected documents are currently accessed by other transactions.

### 3.6 Distribution

As noted in subsection 2.1 multi-user support usually means distribution of the project activities over a number of single-user workstations. These users need to share the projects' internal syntax graph among their workstations. There are basically two ways how this could be achieved. The first is to allow for a distributed access from the users' (client) workstations to a DBSE server. The second way would be to distribute transparently the syntax graph itself over various DBSEs that are locally accessible from user's workstations.

With the first approach, the whole syntax graph resides on the DBSE server. This server would surely become a performance bottleneck for the whole PSDE. Hence, this approach seems feasible only for small projects (say less then 10 users). It is, however, worth consideration, as many projects are either small projects or can be split into fairly independent sub-projects that are small enough.

With the second approach, the process engine can arrange that those parts of the internal syntax graph that represent a particular document are locally accessible from the workstation of the responsible user[4]. The tools that operate on the syntax graph, however, should not need to know anything about the physical distribution of the syntax graph, i.e. the distribution must be transparent for them. It should rather be the responsibility of the DBSE to manage

[4]Though the document itself is locally accessible, there may still be inter-document relationships in the internal syntax graph, that lead to remote nodes.

physical distribution. Following this approach would surely reduce the traffic on the local area network, as well as the amount of expensive remote accesses to the internal syntax graph. Thus much larger projects could be handled following this approach.

### 3.7 DBSE Administration

As the contents of the DBSE might be the most important capital of software houses, issues of data security are very important. This involves two aspects: access to the database and the possibility of hardware crash recoveries.

The DBSE must be able to restrict the access to its objects such that non-authorised persons are excluded from any access. This means that the DBSE must be able to identify its users. Additionally, the DBSE must enforce authentication e.g. by assigning passwords to DBSE users such that it can assure that persons correspond to DBSE users.
Besides this, the access control mechanism required in subsection 3.5 needs users to be known to the DBSE. Thus, for purposes of user management, the DBSE has to offer means to be used by a database administrator (DBA) in order to enter new users and groups to the DBSE, to change user informations like passwords, to remove users from the set of known users and groups from the set of known groups, and to change membership to groups.

Moreover, data stored in the DBSE must be protected from any hardware failures such as disk crashes. Therefore the DBSE has to offer means for dumping the contents of the DBSE to backup media like e.g. tapes. As the size of a project database may be too large to be completely backed up daily, the DBSE must allow for incremental backups. The DBA should not have to shut down the database in order to perform these incremental backups.

### 3.8 Views

As proposed in subsection 3.1, the project graph may contain a lot of redundant information – in figure 2 the nodes *Function*, *Module*, *DeclIdent*, *OperationList* and the edges *ToIdent* and *ToExport* are duplicated in the design and implementation subgraphs. Eliminating this duplication by sharing the aggregation subtrees concerned has the following advantages: (1) the conceptual schema is simplified (2) storage of the schema and corresponding data requires less space, and (3) consistency preservation especially across document boundaries becomes much easier. Such sharing cannot be contemplated, of course, if automatic consistency preservation between documents is inappropriate.

If subtree sharing is to be used, tools accessing the project graph need a view mechanism like that offered in many relational database systems, to maintain appropriate separation of tool concerns and so allow separate tool development and maintenance.

Figure 4 sketches how the redundancy which exists in figure 2 is reduced in a new schema. The schema will be accessed by the design editor and the implementation editor through two views. The figure depicts the conceptual project-graph together with the design and implementation view of this graph.
The **design view** of a function module node is declared to hide the *ToHiddenOps* edge defined in the conceptual schema as these operations shall not be seen at the module interface level. The design view also renames the *ToExportedOps* edge of a conceptual function module node to *ToExport*. Finally, the design view hides the *ToDecl* and *ToStat* edges defined for procedure

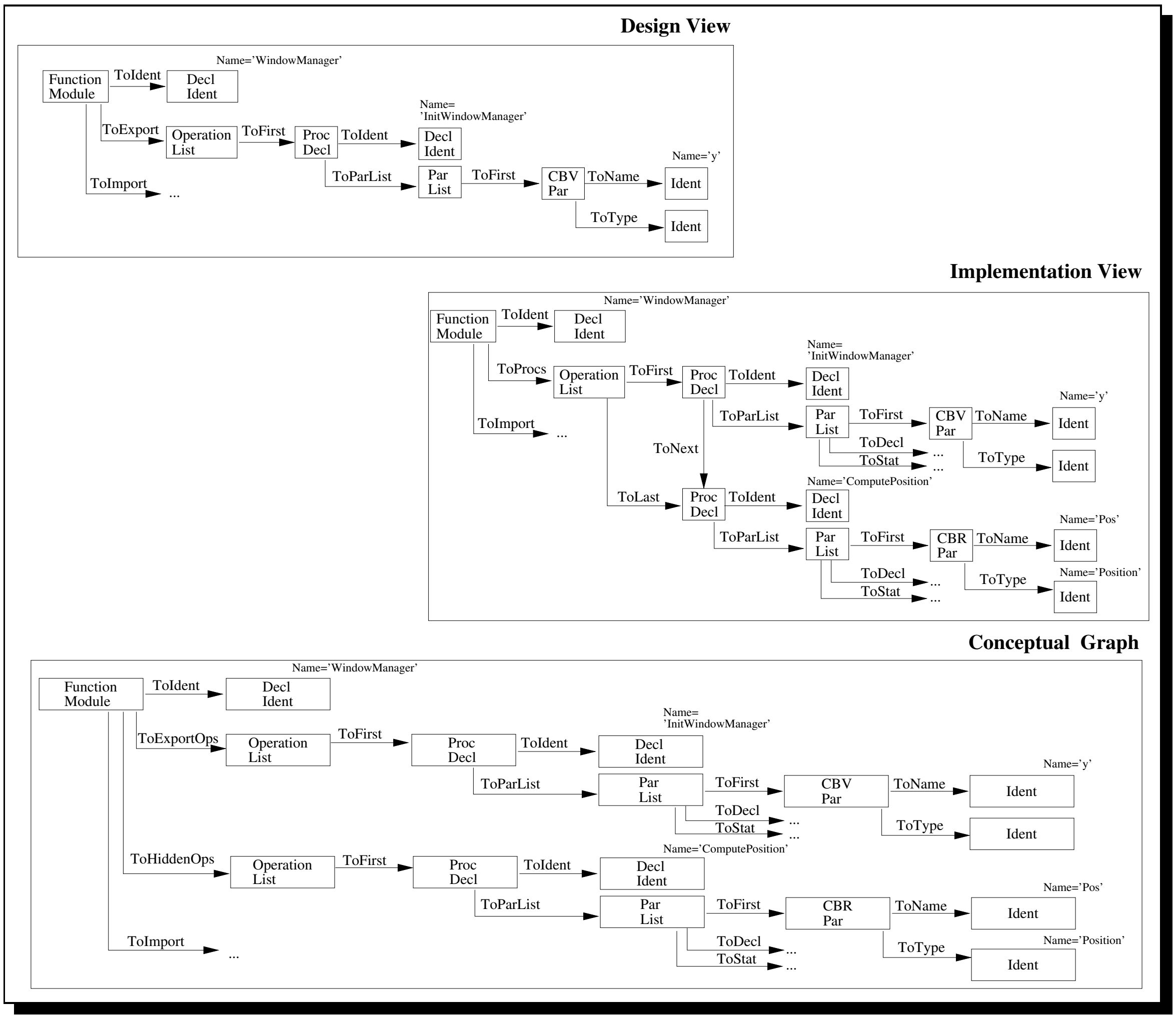


Figure 4: Reducing Redundant Information with Views

declaration nodes in the conceptual schema from each procedure declaration node in the design view. Consequently, all node types reachable only from these edges in the conceptual schema are also hidden in the design view.
The **implementation view** of a function module declares the *ToProcs* edge to lead to a node of a virtual node type which is constructed by concatenation of the lists accessed via the *ToExportedOps* and *ToHiddenOps* edges of a function module in the conceptual schema. This allows us to have exported and hidden operations merged in one operation list of the implementation graph.

These tool-oriented views must be regarded as virtual structures since they are not actually stored in the DBSE, but updates on views by tools must propagate automatically to the underlying project graph.

Introduction of such a view mechanism, however, has non-trivial consequences for the access control mechanisms required (since users see this access control as applying to document views) and for the versioning policy which the PDSE designer or the users may adopt. Detailed resolution of these issues is largely a matter of PSDE design and is thus beyond the scope of this paper. We note, however, that provision of such a view mechanism is clearly a requirement for effective PDSE design and implementation.

# 4 Existing Database Management Systems

## 4.1 Relational Database Management Systems

If syntax graphs are shared in a relational database management system (RDBMS), a table has to be defined for each node type. A tuple in such a table then represents a node. Each table has a column with unique identifiers (UIDs) used for node identification. Edges are then implemented by additional columns of tables, which contain UIDs of the nodes to which edges lead. Additionally, each table may have columns used for storing attribute values of nodes.

The performance of operations which access a document subgraph as a whole (e.g. unparsing or searching) is however very slow. These operations have to execute a query whenever they traverse an edge. A document subgraph often consists of 500 up to some 10,000 nodes and these operations have then to perform as many queries to read the whole document. As a supporting example, consider the results which are reported in [Lin84]. Unparsing 1,000 lines of program source code stored as an abstract syntax graph in INGRES and implemented on a VAX 11/750, took more than 200 minutes.
A second problem is that an RDBMS can not identify those tuples that implement nodes belonging to a document subgraph, because UIDs do not distinguish between aggregation and reference. Hence the RDBMS can never manage different versions or revisions of a document subgraph on its own.

Relational data definition languages also do not completely fulfill the requirements of an appropriate DDL. By declaring a table in a relational DDL, one can declare a node type as well as the number of edges which start from that node type. It is impossible, however, to define that an edge must lead to a particular node type.
Moreover, it is impossible to store arbitrary node attributes because firstly, only predefined attribute types can be used and secondly, attributes are most often restricted in their length[5]. Furthermore, polymorphic structures can not be expressed. Finally, encapsulation of edges by operations as well as inheritance are not at all supported by RDBMSs.

Schema updates can be performed on relational databases without losing any unaffected data. Updates require most often that all applications which operate on the schema have to be changed and recompiled.

Access to the project graph can only be controlled on type level. That is, one can define an owner for a table and the owner can grant read or write access for that table to other users. However, access rights for complete document graphs rather than for all nodes of a particular type can not be defined.
With respect to concurrent access to the project graph RDBMSs offer only conventional transactions with locking transparent to the application. Hence, locking is always performed and the process engine cannot run its own customised transaction schemes.

Distributed access to a relational database as well as database administration must be regarded as sufficient for our purposes. The former is implemented by client accesses to a central database in most RDBMSs. Some even allow distribution of data among databases on different hosts.

RDBMSs offer view definition capabilities. In general, however, these views can not be updated

[5] Oracle 6.xx, for instance, can not store longfield attributes which are longer than 64 KBytes.

in a way that the update migrates into the underlying conceptual project graph. The problem is known as *view-update problem* in literature [GPZ88].

## 4.2 Structurally object-oriented Database Management Systems

Structurally object-oriented databases [Dit86] have been developed during the last decade, some of them specifically to serve as DBSEs. Typical representatives are PCTE/OMS [GMT87], CAIS-A [AJPO88], Damokles [DGL86], PGraphite [WWFT88] and GRAS [LS88]. They differ widely with respect to their data models[6], their programming language interface, their reliability[7] and the granularity of objects which significantly influences the DBSE design[8]. In this subsection, we explain why none of the above mentioned database systems fulfill all of our requirements. For a fully elaborated discussion of this topic, we refer to [DEH+91]. A short version is available as [DEL92].

In PGraphite and in GRAS, efficient storage, retrieval and manipulation of abstract syntax graphs that represent one document cause no problems, as they were especially tailored towards this task. In GRAS, however, a project graph must be structured into subgraphs whose maximum number of nodes is limited. For edges between subgraphs, distinct (low performance) manipulation operations have to be used.
In PCTE/OMS, CAIS-A and Damokles, storage of syntax graphs causes no conceptual problems, as syntax graphs can be considered as kinds of E/R models. From the performance point of view, however, these databases are unsuitable. They suffer from what we call the *granularity problem*. A performance evaluation of PCTE/OMS and Damokles has shown that they have not been designed to cope with large numbers of small objects, because their implementations neglect any kind of object caching and clustering mechanisms that are necessary for an efficient manipulation of large abstract syntax graphs [DEH+91].

Damokles and PCTE/OMS support versioning. In PCTE's VMCS, composite entities may be versioned. Composite entities could implement subgraphs of the internal syntax graph. Then tools could easily create and delete versions of a subgraph and navigate through the version history graph in order to select a current version. In Damokles versioning is implemented similarly, though it is not possible to set a current (preferred) version.
PGraphite and GRAS do not support versioning of subgraphs at all. Neither does CAIS-A support versioning of composite objects.
In none of the systems is it possible to define duplication strategies for nodes or handling strategies for edges.

PCTE/OMS and CAIS-A provide group-oriented discretionary access control to objects. PCTE allows to define a flat group structure, whereas groups in CAIS-A may be nested. In PGraphite, GRAS and Damokles, all objects are accessible by anyone.
PCTE/OMS provides activities (similar to atomic and durable transactions), transactions (what we call conventional transactions) and nested transactions. Transactions in CAIS-A are conventional.
PGraphite and GRAS have no means of concurrency control at all, while the concurrency

[6] PCTE/OMS, CAIS-A and Damokles are based on extended entity-relationship models, whereas PGraphite and GRAS implement persistent storage of graphs where nodes may be attributed, but edges may not.

[7] PCTE/OMS has become a product whereas Damokles, PGraphite and GRAS are more or less research prototypes.

[8] PCTE/OMS, CAIS-A and Damokles have been designed basically to contain large objects and be upwards compatible with file-systems whereas nodes in PGraphite and GRAS can not contain large attributes

control mechanisms in Damokles that provide for conventional and design transactions have never been fully implemented.

Objects and relationships in PCTE/OMS can be transparently distibuted over hosts in a LAN. Moreover, objects in PCTE/OMS may be replicated, i.e. they are physically located on several hosts, though logically seen as one object by the applications, In CAIS-A and Damokles, objects and relationships can be distributed. In CAIS-A the distribution is transparent, but not in Damokles. Moreover, CAIS-A and Damokles do not provide any facilities for transparent object replication.
PGraphite and GRAS neither support distribution of nodes and edges nor allow distributed access to a central database.

Both, PCTE and CAIS-A support views. In PCTE, users or tools can select their working schema as a union of schema definition sets that are known to the OMS. CAIS-A offers a similar mechanism. With these mechanisms, it is possible to hide nodes and edges from particular tools and to rename node types and edges types. In both view mechanisms, updates made through these views migrate to the underlying objectbases.
None of GRAS, PGraphite or Damokles support any kind of view mechanism.

## 4.3 Fully object-oriented Database Management Systems

During the last few years, fully object-oriented database management systems (ooDBMSs) [ABD+90] such as GemStone [BMO+89], $O_2$ [BDK92], Versant [Gor87], Ontos [AHS91] or ITASCA (the former ORION [KBC+89]) have evolved from research prototypes into products. They are intended to be used in standard applications, and in non-standard areas such as CAD, Hypertext and Hypermedia, or CASE. In this subsection we investigate, how they meet our requirements.

To store syntax-graphs in ooDBMSs, one implements nodes by objects, edges by instance variables of objects, node types by classes and types of nodes edges lead to by types/classes of instance variables. Some ooDBMSs distinguish instance variables with aggregation and reference semantics in order to build complex objects (e.g. the ITASCA system). In these ooDBMSs, complex objects can implement document subgraphs.
OoDBMSs enable efficient edge traversal by dereferencing instance variables. Performance evaluations show that some ooDBMSs are at least in single-user mode efficient enough to be used in the way sketched above [EK92].

In most ooDBMSs, the conceptual schema is defined by class definitions given in an object-oriented programming language (e.g. C++ in Versant, Ontos and ObjectStore, and Common-Lisp in ITASCA). Others offer special purpose languages (e.g. OPAL in GemStone and $O_2C$ in $O_2$). In all of them, classes, instance variables and their types can be defined.
Mandatory type constructors for ooDBMSs are tuples, sets and lists [ABD+90]. Many ooDBMSs offer additional constructors such as bags or dictionaries. These constructors may not only be used for defining node types, but also for user-defined attribute types.
As classes inherit from other classes, generic node types can be constructed by a PSDE designer. Inheritance furthermore enables polymorphic aggregations.
Instance variables are encapsulated with methods. They can be used to define graph modifying operations and hence to enforce the integrity of the project graph.
Some ooDBMSs offer a query language and ad-hoc query facilities (e.g. $O_2Query$ in $O_2$).

Most ooDBMSs do not support schema updates. Some allow modification of the schema, but do not migrate data to the new schema versions. Instead, they maintain for each object the version of the schema that was valid at the time the object was created [PS87]. In general, the problem of schema updates as required for DBSEs must be regarded as unsolved for ooDBMSs.

Most ooDBMSs do not offer any versioning mechanisms. Some (e.g. Versant) offer versioning of objects and means for maintaining version histories for single objects. A few (e.g. ITASCA) even offer versioning of complex objects. None of them enables the selection or definition of duplication strategies for objects. Neither does any of the systems offer handling strategies for edges between complex versioned objects.

The extent and form of multi-user support varies significantly from system to system. To discuss all systems w.r.t. multi-user support in an appropriate level of detail is beyond the scope of this paper. Instead we exemplify this aspect by presenting the approach taken in GemStone.
GemStone allows access rights to be defined for objects by declaring them (at run-time) to be members of a particular segment which in turn declares the access rights for all its member objects. A segment has an owner and a number of groups to which the owner may grant read or write access.
GemStone offers an optimistic transaction scheme and enables locks of different modes to be set explicitly. Moreover, conflict detection in optimistic transactions may be influenced by the user such that isolation and consistency preservation properties of transactions may be switched off.

All ooDBMSs offer distributed access to a database by a client/server architecture. There are basically two variations of client/server architecture: client-oriented (e.g. $O_2$) and server-oriented (e.g. GemStone). In the client-oriented architecture, the client performs computation of methods while the server is responsible only for transferring the needed objects to the client. In the server-oriented architecture, methods are executed on the server. We expect that the client-oriented architecture is more tolerant against higher load because much of the computation can be performed on clients.
None of these ooDBMSs is capable of managing distributed databases. As long as this is not the case, ooDBMSs cannot be used in arbitrarily large projects.

To date, there are no ooDBMSs with view definition capabilities. However, there are proposals for views in ooDBMSs [SLT91, AB91] and prototypes are being built. The view definition mechanism defined in [AB91] exactly fits the requirements we presented on views.

# 5 Conclusions and Further Work

Relational DBMSs are inappropriate for storing project graphs, since (1) the data model can not express syntax graphs appropriately, (2) RDBMSs do not support versioning of document subgraphs and (3) RDBMSs can not be used to implement customised transaction schemes.
No structurally object-oriented DBMS meets all of our requirements. Those that are capable of efficiently managing project graphs, lack functionality w.r.t. views, versioning, access rights and adjustable transaction mechanisms, and distribution. Others that offer these functionalities are unable to manage the large collections of small objects as they occur in project graphs and are therefore inappropriate. Moreover, none of these systems enables encapsulation of nodes with operations, which we regard to be crucial.

For this reason we focus on ooDBMSs in the future. As general databases, they have not been designed to manipulate a particular granularity of objects. They provide powerful schema definition languages which particularly enable encapsulation, inheritance and polymorphism. Some have been proven to perform fast enough even while accessing document subgraphs from a remote host [EK92].

Currently, we are porting the Merlin PSDE to an ooDBMS and enhancing Merlin with syntax-directed tools which store their document subgraphs in this ooDBMS. Merlin is a research project (c.f. [PSW92, PS92]) at the University of Dortmund carried out in cooperation with STZ, a local software house. One result of Merlin is the prototype of a PSDE based on a rule based description of the software process
Moreover, we are participating in the ESPRIT-III project GoodStep (General Object-Oriented Database for SofTware Engineering Processes). A major effort in GoodStep is devoted to enhancing the $O_2$ ooDBMS to be particularly suitable as a DBSE. The project will improve version management, add view definition capabilities and break up the transaction management to enable implementation of customised transaction schemes with the $O_2$ system.

### Acknowledgements

We are indebted to the members of the Merlin project, namely G. Junkerman, C. Lücking, O. Neumann, B. Peuschel and S. Wolf, for a lot of fruitful discussions about PSDE architectures. We thank the participants of the GoodStep project for stimulating discussions, in particular, Prof. C. Ghezzi, Prof. R. Zicari, Dr. S. Abiteboul, Dr. P. Armenise, Dr. A. Coen. Last, but not least, we enjoyed working with Prof. U. Kelter, Dr. S. Dewal, F. Buddrus, D. Dong, H. Hormann, M. Kampmann, D. Platz, M. Roschewski and L. Schöpe on the experimental evaluation of existing database systems.
The joint work described here was done at University of Dortmund while J. Welsh was on study leave from University of Queensland. The authors are grateful to the German Ministry of Research and the Australian Department of Industry Trade and Commerce for enabling this cooperation.

## References

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. of the ACM SIGMOD Conf. on Management of Data, Denver, Co*, pages 238–247. ACM Press, 1991.

[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.

[AHS91] T. Andrews, C. Harris, and K. Sinkel. Ontos: A Persistent Database for C++. In R. Gupta and E. Horowitz, editors, *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, pages 387–406. Prentice-Hall, 1991.

[AJPO88] Ada Joint Program Office. Common Ada Programming Support Environment (APSE) Interface Set (CAIS), Revision A. Technical Report DoD-STD-1838A, U.S. Department of Defense, 1988.

[BC85] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. *ACM SIGPLAN Notices*, 20(7):34–42, 1985.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGSOFT Software Engineering Notes*, 13(5):14–24, 1988. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Mass.

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

[Ber87] P. A. Bernstein. Database System Support for Software Engineering. In *Proc. of the $9^{th}$ Int. Conf. on Software Engineering, Monterey, Cal.*, pages 166–178, 1987.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

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

[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.

[DEL92] S. Dewal, W. Emmerich, and K. Lichtinghagen. A Decision Support Method for the Selection of OMSs. In *Proc. of the $2^{nd}$ Int. Conf. on Systems Integration, Morristown, N.J.*, pages 32–40. IEEE Computer Society Press, 1992.

[DGL86] K. R. Dittrich, W. Gotthard, and P. C. Lockemann. Damokles – a database system for software engineering environments. In R. Conradi, T. M. Didriksen, and D. H. Wanvik, editors, *Proc. of an Int. Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 353–371. Springer, 1986.

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

[EK92] W. Emmerich and M. Kampmann. The Merlin OMS Benchmark – Definition, Implementations and Results. Technical Report 65, University of Dortmund, Dept. of Computer Science, Chair for Software Technology, 1992.

[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.

[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.

[Gor87] K. E. Gorlen. An Object/Oriented Class Library for C++ Programs. *Software — Practice and Experience*, 17(12):181–207, 1987.

[GPZ88] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–521, 1988.

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

[JF82] G. F. Johnson and C. N. Fisher. Non-syntactic attribute flow in language based editors. In *Proc. of the* $9^{th}$ *Annual ACM Symposium on Principles of Programming Languages*, pages 185–195. ACM Press, 1982.

[KBC+89] W. Kim, N. Ballou, H.-T. Chou, J. F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 251–282. Addison-Wesley, 1989.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. In *Proc. of the ACM SIGMOD 1985 Int. Conf. on the Management of Data*, pages 388–401. ACM Press, 1985.

[Lin84] M. A. Linton. Implementing Relational Views of Programs. *ACM SIGSOFT Software Engineering Notes*, 9(3):132–140, 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Penn.

[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.

[Mad92] N. H. Madhavji. Environment Evolution: The Prism Model of Changes. *IEEE Transactions on Software Engineering*, 18(5):380–392, 1992.

[NSZ91] M. H. Nodine, A. H. Skarra, and S. B. Zdonik. Synchronization and Recovery in Cooperative Transactions. In *Implementing Persistent Object Bases – Principles and Practice – Proc. of the* $4^{th}$ *Int. Workshop on Persistent Object Systems* , pages 329–342, 1991.

[PKH89] C. Pu, G. Kaiser, and N. Hutchinson. Split transactions for open-ended activites. In *Proc. of the* $14^{th}$ *Int. Conf. on Very Large Databases*, pages 26–37. Morgan Kaufman, 1989.

[PS87] D. J. Penney and J. Stein. Class Modification in the GemStone object-oriented DBMS. *ACM Sigplan Notices*, 22(12), 1987.

[PS92] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proc. of the* $14^{th}$ *Int. Conf. on Software Engineering, Melbourne, Australia*, pages 262–279. IEEE Computer Society Press, 1992.

[PSW92] B. Peuschel, W. Schäfer, and S. Wolf. A Knowledge-based Software Development Environment Supporting Cooperative Work. *International Journal for Software Engineering and Knowledge Engineering*, 2(1):79–106, 1992.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Deductive and*

*Object-Oriented Databases – Proc. of the $2^{nd}$ Int. Conf., DOOD '91, Munich, Germany*, volume 566 of *Lecture Notes in Computer Science*, pages 189–207. Springer, 1991.

[Tho89] Ian Thomas. Tool Integration in the Pact Environment. In *Proc. of the $11^{th}$ Int. Conf. on Software Engineering, Pittsburg, Penn.*, pages 13–22. IEEE Computer Society Press, 1989.

[TSY+88] R. N. Taylor, R. W. Selby, M. Young, F. C. Belz, L. A. Clarce, J. C. Wileden, L. Osterweil, and A. L. Wolf. Foundations of the Arcadia Environment Architecture. *ACM SIGSOFT Software Engineering Notes*, 13(5):1–13, 1988. Proc. of the $4^{th}$ ACM SIGSOFT Symposium on Software Development Environments, Irvine, Cal.

[WBK91] J. Welsh, B. Broom, and D. Kiong. A Design Rational for a Language-based Editor. *Software — Practice and Experience*, 21(9):923–948, 1991.

[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.