1

Information Management in Process-Centered Software Engineering Environments

Naser S. Barghouti,[†] Wolfgang Emmerich,[§] Wilhelm Schäfer[‡] and Andrea Skarra[†]

[†] *AT&T Bell Laboratories, USA*
[§] *University of Dortmund, Germany*
[‡] *Universität Paderborn,Germany*

ABSTRACT

Process-centered software engineering environments (PSEEs) generate and maintain a significant amount of information in their support of large-scale software development. Their architectures typically include a repository that stores product data, process enactment data, or both. Different PSEEs use different types of repository: some use extensions of commercial database technology, some use special-purpose repositories, and others use structured files. To date, however, there has been little work done to distinguish their use of repositories from those of other applications with persistent data. This paper identifies the requirements imposed by PSEEs on the functional capabilities of a repository. It evaluates whether or not existing database technology satisfies these requirements, and it presents our experience in using and enhancing repository technology for PSEEs.

1.1 INTRODUCTION

A major theme in software engineering research has been the design of environments that assist in the development and maintenance of large software systems. Toward this end, process-centered software engineering environments (PSEEs) provide assistance by enacting an explicit model of a project's software development process. The enactment ranges from simply guiding humans through the process to enforcing or actually executing the model.

Process-Centered Environments, Edited by Fuggetta and Wolf
© 1995 John Wiley & Sons Ltd

A PSEE maintains a significant amount of information about the project under development. It stores the information in a repository to maintain consistency and to support queries. In this respect, a PSEE is like a database application. However, its data management requirements differ in two ways from those of applications that use a commercial system as their repository. First, a PSEE maintains data that is quite heterogeneous in form, ranging from highly structured objects (e.g., code, documentation, abstract syntax graphs) to simple data items (e.g., user names, timestamps). Second, a PSEE accesses the data within the context of activities that are long-lived, open-ended, and interactive. The heterogeneity of the data together with the access characteristics impose certain information management requirements that are not satisfied entirely by a single commercial database system.

The objective of this paper is to provide a comprehensive understanding of how PSEEs manage project information. It describes the various forms of data generated and maintained by PSEEs, it characterizes how PSEEs access this data, and it delineates the resulting data management requirements. It also evaluates the currently available commercial database management systems (DBMSs) against these requirements, and it discusses present and future trends in PSEE information management architectures.

The evaluation considers the two most widely-purchased kinds of DBMS today: relational and object-oriented. Our conclusion is that there is no single commercial system that completely satisfies the PSEE data management requirements. Relational DBMSs are a mature technology that provides excellent support for querying, distribution, and the handling of large amounts of data. Moreover, they exhibit very good performance. However, they do not provide sufficient support for maintaining the kinds of data generated by PSEEs. In contrast, object-oriented DBMSs provide excellent support for storing and accessing complex and structured objects, but they are a relatively immature technology with poor support for querying and distribution. Furthermore, most object-oriented DBMSs are tightly coupled to a single programming language, complicating integration with applications that are written in a different programming language.

The paper is organized as follows. Section 1.2 presents a general architecture for a PSEE, concentrating on the information management component. Section 1.3 examines the nature of the information maintained by a PSEE. It explores the data access characteristics, and it delineates the data management capabilities required in the information management component of a PSEE. Section 1.4 evaluates how well existing database systems satisfy the requirements. Section 1.5 describes the present trends in PSEE information management, detailing our experiences in building two PSEEs, Marvel and Merlin, while Section 1.6 describes two future trends illustrated by the GOODSTEP and Provence projects. Section 1.7 concludes the paper with a summary and some overall conclusions about how best the information management component of a PSEE can exploit existing database technology in concert with file system technology.

1.2 THE INFORMATION MANAGEMENT COMPONENT

A typical PSEE architecture consists of at least three components: (1) a multiuser interface, (2) a process support component, and (3) an information management component (Figure 1.1). The multiuser interface supports multiple, possibly distributed, user clients and programs. The process support component (PS) contains one or more process enactment engines, each of which provides one or more process modeling formalisms. All process enactment and process

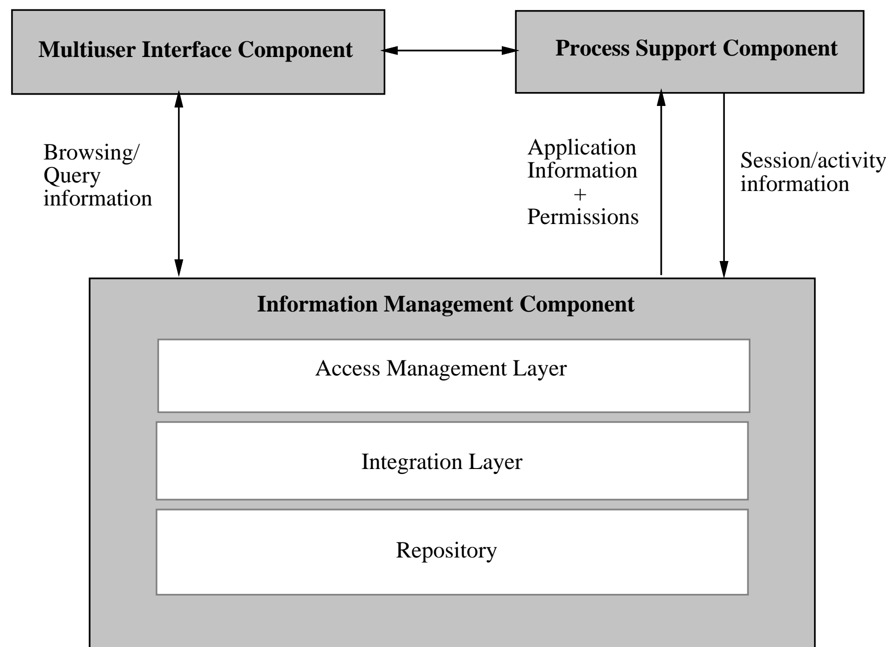

Figure 1.1 The Information Management Component

execution activities pass through the PS. The multiuser interface and the process support component are addressed in detail elsewhere in this issue. This paper concentrates on the information management component, and in particular its functional capabilities, interface to the other components, and possible realizations.

The information management component (IM) is responsible for maintaining the consistency and availability of the information needed by the other PSEE components. It stores this information in a *repository*, by which we mean a container of data. The repository may be composed of multiple database systems and/or collections of files that are managed by the underlying file system. In the case of a heterogeneous repository, there is a need for an *integration layer* to manage activities that span multiple databases, integrity constraints across database boundaries, and so on.

The IM interacts directly with the multiuser interface to allow querying and browsing of the data stored in the repository. The PS communicates information about enactment transition, user requests, and activity executions to the IM. The *access management layer* of the IM uses the information from the PS to create transactions, determine permissions, obtain locks, and so on. It provides the requested information to the PS only if the request does not violate the specification for consistent access to the repository.

An interesting architectural issue is the interface between the IM and the PS. It may be argued that it is unrealistic, or even undesirable, to create a solid boundary between the IM and the PS, especially with regard to transaction management for interteam coordination and intrateam cooperation during process execution. Indeed the architectures of some PSEEs, such as Marvel [12] and Merlin [50], do not clearly separate the functions of the IM from those

of the PS. The reason for this, however, is that at the time when these PSEEs were built, the available database technology did not satisfy their requirements, such as support for long-lived sessions, and thus they had to implement functions in the PS to meet these requirements.

In retrospect, we believe that data management functions, including transaction management, should be handled entirely by the IM, and to do so the IM must satisfy the information management requirements of PSEEs, which we delineate in the next section. This does not preclude the need for constant exchange of information between the IM and the PS. The IM must still obtain information about process model enactment and process execution in order to manage open-ended activities.

1.3 INFORMATION MANAGEMENT REQUIREMENTS

The nature of the software development process imposes a set of requirements on the IM of a PSEE with regard to its functional capabilities. In this section we identify these requirements, concentrating on those requirements that are especially germane to PSEEs and that are incompletely satisfied by current data management systems. Moreover, we disregard requirements that are common to virtually every application with persistent data, such as the requirement for a data definition language and a data manipulation language. The requirements we present here are not necessarily unique to PSEEs; they may apply equally to other design domains.

We suggest the following categorization as a taxonomy of information management requirements for PSEEs. Other taxonomies appear elsewhere [8, 22, 36, 61, 62].

1.3.1 Data Composition and Structure

A software development project typically generates many different forms of data, as illustrated by the following (nonexhaustive) list of common data types:

- *product data*, such as source code, configuration management data, documentation, executables, test suites, testing results, and simulations
- *process data*, such as an explicit definition of a software process model, process enactment state information, data for process analysis and evolution, history data, and project management data
- *organizational data*, such as ownership information for various project components, roles and responsibilities, and resource management data

The boundary between these categories is not always firm. For example, configuration management data, which is part of the product, includes some part of the history of development, which is process data.

In each of the three categories, the data items may be composed and structured in various ways. They may be stored as complex objects (i.e., objects with multiple attributes), composite objects (i.e., objects like modules, libraries, and manuals that contain other objects), flat files in ASCII or binary, pointer-based data structures (e.g., an abstract syntax graph representing a product data object such as a program), contiguous data structures (e.g., arrays), or simple basic data units (e.g., integers or strings).

Moreover, the data items in a software project are densely interconnected by a variety of *relationships*. Examples of relationships include *derivation* (e.g., between an executable object and the source object from which it is compiled), *dependence* or *interobject consistency*

constraints (e.g., between a document and the executable that it describes), *version order* (e.g., between two or more incarnations of an object from a history of its modifications), and *configuration* (e.g., executable objects that are linked into the same system).

The IM in a PSEE must efficiently handle the storage and retrieval of all the data forms. For example, it must implement schemes for disk layout and clustering of data that are commonly accessed together, and it must implement schemes for transforming data items from their main memory representations to their persistent memory representations (and vice versa) while maintaining relationship information among them.

1.3.2 Consistent Access to Data

An important function of the IM is to guarantee consistency in the repository data across repeated and concurrent access. We describe three mechanisms to preserve data integrity: a well-defined abstract interface to the data, transaction processing, and support for constraints and triggers.

1.3.2.1 Data encapsulation

A well-established means of preserving data integrity is the technique of data encapsulation: each data type specification includes a well-defined functional interface for instances of the type. An object cannot be accessed by a program or by another object except by way of its interface functions, commonly known as methods. The methods implement consistent access to the object. For example, a class *Symbol_Table* may define a method *Insert* that adds a new entry only if the key of that entry is not already in the table. The method implements the consistency constraint that keys in symbol tables are unique.

Data encapsulation preserves consistency at the granularity of a method (i.e., for a single access to a single object). To guarantee consistency at a coarser granularity (i.e., across multiple accesses to one or more objects), a larger unit of synchronization is required, such as a transaction.

1.3.2.2 Transactions and sessions

In traditional data management systems, a transaction is the unit of interaction between an application and the system. It consists of a series of accesses to the database together with some computation. The systems tend to be optimized toward supporting transactions with the following characteristics:

- *Short duration of execution and limited scope*
  Computationally each transaction is fairly simple, and it accesses only a small portion of the data in the repository.
- *Completeness*
  Each transaction is a logical unit of work that represents a single task in its entirety.
- *Consistency*
  A transaction maintains global consistency in the database (i.e., each transaction satisfies every consistency constraint defined on the data). It takes a database from any consistent state to another consistent state.

- *Independence*
  Transactions are defined incrementally and independently of one another. The data management system preserves their independence during execution by guaranteeing the isolation of each.

A traditional data management system preserves data consistency by guaranteeing the execution atomicity, consistency, isolation, and durability of transactions. These properties are often referred to as the ACID properties of transactions.

In contrast, the software engineering domain supported by PSEEs is characterized by complex, long-lived, and interactive tasks. To manage and reduce the complexity, the tasks are usually divided into simpler, parallel subtasks that preserve design modularity. The subtasks are distributed together with their associated data among people and machines. As a result, the interaction between a software developer and a PSEE is quite different from the transactional interaction between the user of a banking application, for example, and a database management system. To distinguish the developer style of interaction, we use the term *session* instead of transaction for the unit of interaction with a PSEE. Sessions differ from transactions in the following ways:

- *Long-lived, dynamically-defined, and interactive execution*
  A session is a lengthy activity that is typically interactive (e.g., an edit and debugging session). Moreover, a session is commonly open-ended and dynamically-defined; the actions it performs and the objects it accesses cannot be determined *a priori*.
- *Partial completeness*
  A session typically performs only part of a task. The task is complete only when all the sessions that correspond to the task's subtasks finish successfully.
- *Partial consistency*
  Subtasks are commonly spanned by data constraints. That is, the data sets associated with the sessions corresponding to the subtasks are interrelated by constraints. As a result, a single session may leave the repository in a partially consistent state by failing to satisfy some constraint. For individual sessions to effect global consistency, each must at least check the state of all data that are transitively related to the session's original data set, resulting in its accessing an unreasonably large amount of data due to the complexity of the data relationships. Instead, a PSEE repository reaches global consistency incrementally as the sessions corresponding to the task's subtasks complete.
- *Interdependence and collaboration*
  Sessions are interdependent with respect to both the data they access and the task of which they are subtasks. That is, they depend on each other for correctness and completeness of the entire task. They must share data collaboratively (i.e., in ways that violate their isolation) due to the complexity of the corresponding subtasks, the length of their execution, and the interrelationships among their data. As a result, they develop data dependencies on each other while executing, such as when one session reads data items that were written by another uncommitted session.

PSEEs need a flexible, customizable session management mechanism that can be integrated with configuration and version management. The PSEE should implement a mechanism by which a project can specify the correct execution of its sessions, due to their partial completeness and consistency. Current data management systems do not provide adequate facilities for this kind of specification.

1.3.2.3 Constraints and triggers

To facilitate reasoning about the behavior of a system and to reduce redundancy in the code, it is useful for the system to support a machine-readable language for constraint specifications. Given such a language, a data type's constraints can be defined all together in a location separate from the type's data definition and methods. The constraint checking code does not have to be repeated in every method on the type; the system automatically checks the constraints upon a relevant access. Similarly, interobject constraints can be defined separately from the transaction code and need not be repeated within every transaction.

A general mechanism for enforcing constraint maintenance is a facility for triggers. A *trigger* is a three-tuple $< event\ \ condition\ \ action >$, where the *action* executes upon the occurrence of the *event* if the *condition* holds. For example, given a constraint c on attribute a of object o, a trigger might be $< change_in_a\ \ c_is_violated\ \ abort_transaction >$.[1]

A triggering and notification mechanism is also useful for coordinating the collaborative development of a software project. For example, a triggering mechanism can be used to send notifications to a developer whenever another developer performs a specific action that affects the first developer's work.

1.3.3 Abstraction and Views

A schema describes the data in a repository at a fine granularity and with great detail. The complete description of data is often referred to as the conceptual model. It is not always desirable for all users to see the entire conceptual model of data. For example, some users may be restricted from accessing certain parts of the data due to security considerations. A facility that supports abstract views of the data is required. An abstract view hides certain details of data, and provides a layer for accessing only the part of the data that is visible under the abstraction. Different views on the same data allow the data to be interpreted under different abstractions to better support different kinds of users.

Consider, for example, a library system database that stores information about books and their loan status. A librarian's view of the data may allow the librarian to change the loan status of a book, whereas a customer's view may not allow the customer to access that part of the database.

1.3.4 Evolution

If there is a constant that is common to every software system, it is this: the system is bound to evolve. Managing change is primarily a problem of managing dependencies. When some part of a system is modified, the dependent parts of the system that are affected by the change must be identified, located, and modified to keep the system as a whole consistent.

There are two kinds of change that a PSEE must manage: changes in the application data and changes in the schema (meta data). In the first case, the change is usually localized. For example, a developer may check out a module, modify it, test the modified version, and check it back in. Change management in this case primarily aims to support parallelism and private workspaces, allowing multiple developers to work on the same module or related modules simultaneously. Common techniques for the application to use include designing the product as a whole to be made up of small, independent components so as to minimize check-out conflicts

[1] Note that state-based triggers are a special case of event-based triggers, where the event is "update" implicitly.

among developers. The most common mechanism with which a PSEE supports simultaneous development is a versioning and configuration management facility [72, 74]. The IM should be able to support these change management mechanisms.

In the second case (schema evolution), a change occurs in the *definition* of an object's composition, constraints, and methods. That is, there is a change in the object's type definition. A type change has potentially widespread consequences, affecting not only all instances of the type but also all programs and other types that use the changed type. For example, if an attribute of a type is changed from integer-valued to real-valued, old programs (i.e., those written against the integer-valued attribute) may not be compatible with new instances of the type (i.e., those created with the real-valued attribute), and new programs may not be compatible with old instances of the type.

The PSEE should provide facilities for schema evolution. Current approaches to the problem include those that are based on automatic conversion [14, 17], delayed or lazy conversion [39], and versioning of the schema [68]. In the conversion approaches, all the type's instances must be converted (eventually), and all the affected programs and methods of other types must be located and recompiled against the new type definition. In the versioning approach, each object indicates the schema *version* of which it is an instance, and programs interpret the object according to that version of the type. A versioning approach obviates the need to recompile programs that use a changed type, but it requires the programs to dynamically bind objects to their type definitions rather than statically bind them during compilation.

1.3.5 Distribution

Typically, PSEE users are on different workstations, and they are connected by either a local area network (e.g., Ethernet) or a wide area network (e.g., Internet). In the former case, users are likely to share data via a local server under some concurrency control and caching protocol. In the latter case, however, a user's working data set may be distributed, replicated, or cached at his or her local machine for performance reasons. The IM must provide support for distribution, such as management of distributed sessions, location transparency, and a protocol for data replication and/or caching.

1.3.6 Tool Integration and Interoperability

The aim of a PSEE is to support the development of large-scale software. Currently, there are numerous tools in widespread use that automate parts of the software development process and manage information. Many organizations integrate sets of these tools into working environments. An organization may store its product data in files, for example, and access its data with tools like editors, compilers, debuggers, formatters, and viewers. In addition, the organization may use a variety of programming languages to implement applications that operate on the product data. Further, the organization may encapsulate some of its process and organizational data within tools, such as project management tools, commercial database systems, or spreadsheet programs, whose data access interfaces are more structured and restrictive than is that of the file system.

Software engineers who have become accustomed to a specific working environment resist changing to a completely new environment, especially before the advantages have been demonstrated conclusively. Therefore, PSEEs should incorporate existing tools and applications rather than supplant them. Toward this end, a PSEE should not follow the closed world

view of existing data management systems in which all relevant data must be in a single database and must be accessed only via the database management system routines.

A more viable PSEE architecture is one in which the repository may incorporate a combination of different storage systems, including files and either commercial or proprietary databases, and in which the IM provides an extensible interface that facilitates integration of existing and new software tools. An open architecture can therefore accommodate data that are not in a single location under the control of a single tool, and it can interoperate with applications written in a variety of programming languages.

1.4 EVALUATION OF AVAILABLE DATABASE SYSTEMS

Two types of database systems are widely available commercially: relational database management systems and object database systems. More recently, a few hybrid object-relational database systems have been developed to address the limitations of the relational and object database systems. In this section, we evaluate each type for applicability to the IM, according to the requirements in Section 1.3. In the evaluation, we distinguish between the problems that are intrinsic in the models behind each type of database system and the missing features that can be potentially added.

1.4.1 Relational Database Management Systems

Relational database management systems (RDBMSs) are the most widely-purchased database technology today. The relational model was developed by Codd in 1970 [27]. Codd's relational model has three basic tenets: (1) it describes data and its manipulation in a non-procedural fashion (relations and queries); (2) it provides a mathematical basis for manipulating relations and maintaining consistency (relational algebra and calculus); and (3) it provides independence of the logical data representation from the physical data representation.

Since 1970, the model has been developed and improved, and numerous RDBMSs have been built. Perhaps the three most important developments as far as this paper is concerned are the following:

1. The adoption of SQL (Structured Query Language), a non-procedural approximation of the relational algebra, as the basis for almost all RDBMS products.
2. The development of the transaction concept and the theory of serializability.
3. The emergence of the client/server computing model and distributed database systems.

Our evaluation in the rest of this section considers not only the basic relational model, but also these three major developments. In addition, it considers the advanced features and extensions provided by the most popular commercial RDBMS products, namely Informix, Sybase, DB2 and Oracle.

1.4.1.1 Data composition and structure

RDBMSs were designed to support one form of data, namely the *relation* (or *table* in SQL). A relation defines one or more typed *attributes* (*columns* in SQL). Data is stored in a table in *tuples* (or *rows* in SQL). Every form of data stored in an RDBMS must be transformed into one or more tables (e.g., see [58] for a discussion about how to store programs in an RDBMS).

Tuples in these tables represent the instances of the corresponding form of data. This points out a fairly obvious shortcoming of RDBMSs: it is often non-intuitive, and sometimes impossible, to model non-tabular forms of data in an RDBMS by hand-coding them on top of the relational model. This is especially cumbersome for PSEEs that support a number of different languages because of the heterogeneity of data maintained by PSEEs, leading to a proliferation of types that must be transformed to the relational model.

Consider storing a complex data object, such as an abstract syntax graphs (ASG), which is used as the common internal representation of a set of syntax-directed tools in an existing PSEE [37, 36]. Like other graphs, an ASG is composed of a set of nodes and edges. Each node has an identity and a type that defines the attributes associated with the node. The attributes can be either flat (e.g., an integer) or structured (e.g., an error list). Edges can be one-to-one (from a single node to another node), one-to-many (from one node to an arbitrary number of nodes, all of the same type), or heterogeneous (from one node to an arbitrary number of nodes of different types).

To store an ASG in an RDBMS, a table must be defined for each node type in the ASG. A tuple in a table represents a node of the respective node type. Each table must have a column for the key attributes that uniquely identify each node. Each flat node attribute is implemented by a column in the type table. Structured attributes require additional tables and references between these tables, because elements in relational tables can store only non-structured values.

One-to-one edges are implemented by additional columns in either the table corresponding to the type of the source node or the table corresponding to the type of the destination node. If an edge exists between two nodes, the node identifier of the source or target node is stored in a column of the tuple implementing the target or source node, respectively. It is not necessary to store node identifiers in both tables to implement an edge. Navigation in both directions is possible because of the associative nature of relational query languages.

A one-to-many edge that originates from a node of type T_1 to an arbitrary number of nodes of type T_2 is implemented by an additional column in the table corresponding to T_2. The table may thus contain multiple tuples with the same value v in this column, meaning that these tuples implement nodes that are all connected to the node identified by v. Implementing one-to-many edges in the table representing the type of the originating node (T_1) denormalizes the table, because the result would be multiple tuples that are identical except for a non-key column that represents the edge.

Heterogeneous edges cannot be implemented in a straight-forward manner because one column cannot reference node identifiers from arbitrary tables. To implement heterogeneous edges, we have to store different types of nodes in one table and store type information in an additional column. However, this leads to many useless elements in that column.

A translation like the one described above causes a serious performance problem. Accesses to an object (especially one that requires pointer chasing) may require several `JOIN` operations on the tables that store the various pieces of the object. A `JOIN` operation in the relational algebra combines data from two tables based on the relationship between the data in those tables. This operation requires scanning of the two tables involved and executing computations, and thus it is an expensive operation. Since access to different parts of an object, including subobjects, are quite common in PSEEs, many `JOIN` operations may have to be executed, causing the performance of the IM to suffer.

Recently, RDBMS vendors enhanced their products with support for a limited form of user-defined types. For example, Sybase, Informix and Oracle can now store large objects, such as images, in the database. These are sometimes referred to as *B*inary *L*arge *OB*jects or

*BLOB*s. However, the internal structure of these objects cannot be defined or manipulated by RDBMS built-in functions; they are simply stored as an uninterpreted sequence of bytes. The implementation of sophisticated functions that manipulate the structure and contents of these fields is still left to the RDBMS user (or administrator), and they are considered external to the database system. BLOBs are thus insufficient for storing complex objects like ASGs in a relational database.

1.4.1.2 Consistent access to data

Data encapsulation

RDBMSs do not support data encapsulation directly. Arbitrary operations that define the functional interface of a data type cannot be defined wholly within the schema using SQL, because it is not computationally complete. For example, one cannot write an SQL query to generate reports such as "print the total of all average sales by agent" from a sales database, because this requires iterating over the result of a query. Operations that cannot be expressed in SQL must be implemented in a host programming language with *embedded SQL* [60]. Embedded SQL allows SQL queries to be included in a program written in another programming language like C or Pascal. To enforce encapsulation of the data type, the schema must define a view for the data type with the operations written in the host programming language as parameters. Access control mechanisms must then be employed to ensure that applications access instances of the data type only through the view.

This method of supporting a limited form of encapsulation cannot be used for user-defined types like BLOBs, since these cannot be passed as parameters to views. Therefore, it is not possible in RDBMSs in general to implement data encapsulation of arbitrary data types. This is one major limitation of current RDBMSs.

Constraints and triggers

Most RDBMSs support some form of triggers (for example, POSTGRES [71] provides rule-based triggers). The events that can be specified as initiators of the triggers are updates to, deletions of or insertions of tuples or attributes. The actions to be executed, however, are in general SQL statements. Consequently, RDBMS triggers suffer from the weaknesses of SQL, namely being computationally incomplete. However, the most recent products from Sybase, Oracle and Ingres support some procedural extensions to SQL in specifying triggered actions.

Transactions and sessions

All RDBMSs support traditional transactions with ACID properties. Most existing RDBMSs implement a variant of either two-phase locking [38] or optimistic concurrency control [54] as their concurrency control protocol to maintain serializability. Recently, several RDBMSs have added some limited mechanisms for relaxing traditional transactions. For example, Sybase allows three levels of transaction isolation, the weakest of which allows "weak reads" (i.e., non-repeatable read operations or read operations that are not entirely transaction-consistent). These mechanisms support cooperation only in a limited sense; in general, it remains true that the transaction support offered by RDBMSs is inadequate for PSEEs.

1.4.1.3 Abstraction and views

RDBMSs offer a view mechanism for defining virtual tables based on already defined tables, referred to as base tables. These virtual tables are not physically stored in the database, but computed based on existing tables when needed. Some RDBMSs support materialized views that are physically stored. As soon as any of the base tables accessed by the query is updated, the view is re-evaluated transparently and incrementally.

This view mechanism can be used to provide multiple views in a PSEE repository, based on levels of abstraction. The information access layer can define a view as a query that projects particular attributes, hiding other attributes and links. However, the use of the view facility in RDBMSs is limited by the view update problem [6, 46]. An update is permitted within a view only if it can be determined which rows of the base tables need to be updated as a result of the view update. This is not always possible for columns in a view that are derived from multiple columns in multiple base tables. This problem is not unique to RDBMSs but it exists in other kinds of databases as well.

1.4.1.4 Evolution

We consider evolution of both the data model (the schema) and the product. RDBMS schemas can be updated by inserting new columns into table definitions or by deleting tables. In case of a newly created column, the attributes in that column are initialized, but the rest of the table is unaffected. Insertion of new columns and deleting tables are sufficient to implement changes in the part of the data model that defines the data composition and structure. However, the integrity of existing tables that depend in some way on the deleted or modified table cannot be checked completely automatically. Instead, data integrity can be re-established through the use of constraints and triggers.

A problem in RDBMSs is the lack of facilities for maintaining consistency between data structure definitions in the schema and operation definitions in a host programming language with embedded SQL. This must be checked and re-established manually whenever the schema is changed. It could, for instance, be the case that a deleted table or column is still used in an embedded query; the RDBMS would not be able to detect this discrepancy except when the query is executed.

The other aspect of evolution is evolution of the application data. The relational model does not directly support most of the forms of data maintained by a PSEE, and thus RDBMSs do not offer any of the automatic versioning operations required for the forms of data discussed in Section 1.3. Instead, versioning and other aspects of application data evolution must be implemented within a host programming language.

1.4.1.5 Distribution

Distribution includes two aspects: the ability to access data from remote sites on a network and the ability to store data on multiple sites. Commercial RDBMSs in general support both.

All widely-used, commercially-available RDBMS products support a client/server architecture in which queries are executed on the server. The architectures are server-oriented, and allow application data managed by an RDBMS to be accessed and modified from different workstations distributed on a network.

Most RDBMSs also support distribution of data. Consistent access to this data within trans-

actions is achieved through some implementation of the two-phase commit protocol, which guarantees the consistent coordinated commit of all the parts of a distributed transaction [21]. For example, version 7.0 of the Oracle RDBMS allows applications to access remote data transparently within distributed transactions.

1.4.1.6 Tool Integration and interoperability

RDBMSs provide facilities for interoperating with other RDBMSs; they also offer a low-level programming interface for searching indexes, fetching pages and extracting tuple information from the database. The ability to embed SQL in host programming languages also provides a mechanism for enveloping external tools so that they can be invoked from within the database.

However, a more complete integration of tools, whereby tools can exchange information with the RDBMS, requires that those tools be rewritten to include embedded SQL statements. This may not be possible if the source code of a tool is not available. In addition, the difficulty of modeling complex data forms in RDBMSs, as explained above, may complicate integration of tools that maintain structured data. For example, consider a graphical documentation tool that keeps track of documents structured in chapters, sections, subsections, and so on. The tool must be able to retrieve and display any part of the document upon request by the user. The structure of the documents must therefore be maintained for efficient retrieval. Integrating such a tool with an RDBMS requires storing the structure of the documents in the RDBMS, which is cumbersome and may require expensive `JOIN` operations, as illustrated before.

1.4.1.7 Summary of RDBMS evaluation

The major result of our evaluation of RDBMSs is that two of the requirements of Section 1.3 present an intrinsic problem to exiting RDBMSs: supporting user-defined data composition and structure, and data encapsulation. The other requirements may not be satisfied completely by commercially-available RDBMSs, due in many respects to the limitations of the current SQL standard, SQL-89 [76, 60], on which most existing RDBMS products are based. However, there is nothing intrinsic in the relational model that prohibits the extension of RDBMSs to satisfy these requirements. In fact, RDBMS vendors have added many features in the past few years that make it possible, although not always simple or elegant, to satisfy most of the requirements. Furthermore, the SQL language standard has been revised more than once to overcome some of the limitations; it is only a matter of time before RDBMS vendors become compatible to the latest SQL standard.

A revised SQL standard was published by the American National Standard Institute (ANSI) in 1992 (known as SQL-92 or SQL2). SQL-92 improved on SQL-89 in several ways, most notably in the areas of support for more advanced schema evolution and transaction consistency levels. The SQL-92 standard defines three levels of compliance, the most complete of which is not available in any product today. Nevertheless, most RDBMS vendors are striving to become fully SQL-92 compliant in the near future, rendering them more suitable for PSEEs.

SQL-92, although a definite improvement over SQL-89, does not address some of our most important criticisms. Most recently, a draft of the newest standard, known as SQL3, was published. SQL3 contains substantial changes to SQL, including the ability to support user-defined abstract data types and data encapsulation. Importantly, therefore, SQL3 overcomes most of the limitations that our evaluation points out.

1.4.2 Object Database Systems

To address the limitations of RDBMSs, a new class of database systems was proposed in the early eighties. This class of database systems combines object-oriented programming languages with database technology, and thus is referred to as *object database systems* (ODBS) [26, 4]. Several ODBSs are now available as commercial products. The most popular ones are O_2 [56], ObjectStore [55], ITASCA [52], GemStone [30], Ontos [3], and Versant [45].

Several characteristics distinguish ODBSs from RDBMSs. Each object in an ODBS has a unique *identity*. An object has a set of *attributes*, some of which may be complex (e.g., set-valued attributes), allowing an object to contain subobjects and form a composite object hierarchy. An object is *encapsulated* by a set of methods, so that the object's attributes can be accessed only through its methods. The structure and methods of various objects are defined in *classes*. The structure and methods may be *inherited* from superclasses. Dynamic binding is used to determine which method to use for executing a particular operation. The language for method definitions is computationally complete (e.g., C++ and Smalltalk). The class hierarchy can be extended by user-defined classes.

A subclass of ODBSs, often referred to as *structurally object-oriented database systems* (SODBSs), provides only a subset of the features mentioned above [31]. In particular, SODBSs do not support full data encapsulation, but rather emphasize the ability to model complex structural aspects of objects. For example, there is a subclass of SODBSs that focus on the efficient management of graphs that consist of attributed nodes and edges, such as GRAS [24], PGraphite [75], Cactis [48] and Adage [42]. A second subclass supports data models based on extended entity-relationship models. Representatives of this subclass include Damokles [32], DASDBS [66], PCTE/OMS [41], PCTE+ [33], ECMA-PCTE, CAIS [2] and PCIS [69]. Since SODBSs do not offer any advantages over fully object-oriented database systems, they are not evaluated as a separate category.

A problem of ODBSs is the lack of uniformity and standards. Indeed, existing ODBSs vary significantly in their details. Rather than evaluate several ODBSs, we have chosen to evaluate O_2 as a representative of ODBSs for two main reasons. First, O_2 has been extended specifically for use by PSEEs as part of the GOODSTEP project [44], as discussed in Section 1.6.1. Second, O_2 Technology, the company that produces O_2, is an active participant in the Object Database Management Group, a special interest group of the Object Management Group (OMG) [26], and its implementation adheres closely to the published ODMG standard, which is expected to emerge as the standard for ODBSs. Whenever appropriate, however, we mention other ODBSs when they differ from O_2 in significant ways.

1.4.2.1 Data composition and structure

ODBSs typically provide a computationally-complete language for data modeling. For instance, O_2 classes are defined in the O_2C data definition language, an object-oriented extension to the C Programming Language. Most other ODBSs are based on C++, with the exception of GemStone, which in addition supports a Smalltalk-based language called *OPAL*. These languages provide several capabilities that facilitate data modeling. These include multiple inheritance, type checking, polymorphism, and schema structuring. These capabilities make ODBSs very suitable for defining any of the forms of data described in Section 1.3. A specific form of data can be implemented as a class, and each object of that form can be implemented as

an instance of the class. Complex and composite objects can be readily defined. Heterogeneous edges in ASGs, for example, can be implemented using polymorphism.

1.4.2.2 Consistent access to data

Data encapsulation

Object encapsulation is one of the defining features of ODBS. Thus it is supported by all ODBSs. Unlike in RDBMSs, data encapsulation is a fundamental concept in ODBSs that is supported directly by the database system.

Constraints and triggers

Like RDBMSs, most ODBSs provide some form of triggering mechanism. In O_2, triggers are implemented as production rules that become part of an O_2 schema [28]. The rules are based on the Event-Condition-Action (ECA) formalism. The event part specifies the events that trigger the rule. Proposed event types are divided into two categories: (1) entity manipulation event types generated by manipulations (creation, deletion, update, etc.) of entities, and (2) applicative event types, which are associated to the begin or end of a transaction, an O_2 program or an application. The condition part is composed of predicates (OQL queries) over entities. The action part is O_2C code. Conditions and actions can operate on persistent or transient entities.

Transactions and sessions

All ODBSs support conventional transactions with ACID properties. Many ODBSs provide mechanisms for statically defining transactions that turn off one or more of the ACID properties. In GemStone, for instance, the administrator can decide to turn off atomicity, allowing parts of a transaction to commit. In O_2, the administrator can decide that databases are accessed without logging, i.e., the atomicity and durability properties are abandoned. These mechanisms provide some of the flexibility required by PSEEs.

1.4.2.3 Abstraction and views

Various view definition facilities for ODBSs [67, 23, 1, 47] have been suggested recently. They all enable definition of different interfaces for the same objects. Like in RDBMSs, the view mechanism in most ODBSs suffers from the view update problem. However, O_2 attempts to overcome this problem with its view mechanism.

O_2 implements the view mechanism proposed in [1, 64]. The mechanism allows a tool builder to specify virtual schemas and virtual databases. A virtual schema definition is based on a root schema; it hides particular classes defined in the root schema and can modify the interface of classes defined in the root schema by creating virtual classes from root classes. An object that is an instance of a particular class and that is accessed through a virtual schema is represented according to the virtual class definition in the virtual schema. Therefore, the virtual schema definition implicitly defines virtual databases. Virtual databases that are instantiated from virtual schemas can be updated. This is possible because a virtual object has the same object identity as the real object from which it is derived.

The view mechanism of O_2 is particularly suitable for a tool builder to define different views of the application data. The overall structure of the application data can be defined in a conceptual schema using O_2's schema definition language. Based on this schema a number of virtual schemas can be defined for tools, so that each tool is provided with its own view of the application data. For a class that implements a node type in the conceptual schema, the virtual schema for a tool includes a virtual class that shows only those edges that are of concern for the tool and hide any others. Node types that must not be seen at all can be hidden by not defining a virtual class for this class. Moreover, the virtual schema can hide those methods that implement modifications that ought not to be invoked by a tool. It can add additional methods that have not been defined in the conceptual schema in order to implement different unparsing schemes or different parsers for different tools, for example.

1.4.2.4 Evolution

Most ODBSs support incremental updates to an already established schema. Only ObjectStore, Objectivity, Versant and O_2, however, enable migration of an existing database to a changed schema. Of these systems, the O_2 ODBS offers the most sophisticated support for controlling migration after a schema update [39].

Although most ODBSs support evolution of the schema, only few support evolution of the data by providing version and configuration management capabilities. ObjectStore and Versant support versions of objects by providing a predefined class from which other classes can inherit the property of being versioned. This can be used to maintain versions of single objects but is of very limited use for the implementation of versions of configurations. Only ITASCA and O_2 provide support for this. In ITASCA, composite objects are defined statically within the schema whereas in O_2 composite objects are determined in a more flexible way at run-time by including objects in a versionable container object.

O_2's version manager provides a pre-defined class `Version` [59]. An object of class `Version` acts as a container for a set of objects that are together under version control. The class includes data structures for maintaining the version history of the composite object and methods for navigating through the version history graph. `Version` offers methods to add or remove objects to or from the composite object. As soon as an object is added to an object of class `Version`, this object is under version control. Moreover, `Version` offers methods to set the *current* version of a container, to *derive* versions from the current version and to set *default* versions, i.e. to determine which version of an object is used if other objects do not address a particular version.

The class `Version` can be used to implement versions of composite objects, such as subgraphs of an ASG [25]. This is done as follows. In the implementation of each node of the abstract syntax graph that represents a document (document node), an instance variable of class `Version` can be added. Then, whenever a node is created, the object implementing the node is added to the container object of the document node to which the node belongs. When a node is deleted, the node is removed from the container object. To derive a new version of a document, the implementation of the document node only calls the derive method on the container object. A similar strategy is chosen for navigation through the version graph of a document, establishing current and default versions, and so on.

1.4.2.5 Distribution

All ODBSs offer distributed access to a database through a client/server architecture. There are basically two variations of client/server architectures: client-oriented (e.g. O_2 and ObjectStore) 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 or the pages on which objects reside to the client. In the server-oriented architecture, methods are executed on the server. In experimentations with both, the client-oriented and the server-oriented architecture, we found that the client-oriented architecture is more tolerant against higher load. This is because most of the computation load for executing abstract syntax graph operations is taken by the clients.

Most ODBSs do not support distribution of data in a way that is transparent to the PSEE. As long as this is not the case, ODBSs cannot be used in large projects that are distributed by nature, such as those involving off-shore development of software.

1.4.2.6 Tool Integration and interoperability

Since many ODBSs can be accessed through a C++ interface, it is not difficult to build tools that interact with the database. O_2, for example, offers a programming interface for embedding object-oriented queries defined in OQL into host programming languages like C or C++. O_2 also provides a programming interface to the O_2 engine, composed of the schema and object manager. This low-level interface can be used to navigate through object hierarchies stored in the database. But this requires changing the tools to use this interface rather than their own main memory representation. At worst, this is unrealistic for tools supplied by other vendors; at best, it is quite tedious.

1.4.2.7 Summary of evaluation of ODBSs

Although ODBSs overcome the two main limitations of RDBMS, namely support for user-defined types and data encapsulation, they suffer from two shortcomings themselves [51]. The first is the lack of a standard query language with the simplicity of SQL and the resulting optimizations that improve the performance of RDBMSs. The second shortcoming is the tight coupling of most ODBSs to a single object-oriented programming language, most commonly C++, which complicates integration of applications written in other programming languages. These shortcomings are indicative of the relative immaturity of ODBSs as compared to RDBMSs. Nevertheless, there are no intrinsic problems in ODBSs that prevent future products from satisfying most of our requirements. In fact, recent ODBS products provide support for multiple programming languages, and some support an object-oriented version of SQL.

Recently, the SPADE group has published a study in which they evaluate the use of three ODBSs in the SPADE PSEE, O_2, Gemstone and ODE [8, 7]. Their evaluation reaches similar conclusions as ours.

1.4.3 Hybrid Database Systems

In response to the acknowledged deficiencies of the commercial relational and object-oriented systems, at least two vendors have developed a hybrid solution, an *object-relational system*,

that combines the good features of both [70, 51]. The basic idea of this approach is to extend RDBMSs with support for some of the more useful features of object-oriented programming.

The Illustra DBMS (formerly Montage [73]) supports general extensibility with user-defined data types that are integrated with the DBMS built-in data types, allowing optimized access to them via either pointer reference or associative retrieval. Illustra supports encapsulation, inheritance, and polymorphism, as well as standard SQL-89, standard security features, data integrity maintenance, and transactions and recovery.

Illustra's architecture is based on the client-server model. New data types are defined by means of object libraries, called DataBlade^{TM} modules, that are linked to the Illustra server. This yields an architecture that supports query-shipping: the Illustra server executes the query on behalf of the transaction created by a client, resulting in reduced communication overhead. Almost all commercial RDBMS products are based on a query-shipping architecture. In contrast, commercial ODBSs implement data-shipping, where the server sends objects to the clients for processing of queries. DataBlade modules are defined by Illustra, third-party developers, or end-user developers. Illustra provides several such modules for types like text, image, time series, and spatial data.

The UniSQL/X system supports development in either a conventional programming language, such as C, or an object-oriented language, such as C++ or Smalltalk. It supports fully a superset of the Core Object Model adopted by the Object Management Group consortium, including encapsulation of data and methods, object identity, multiple inheritance, arbitrary data types, and nested objects. Moreover, it supports the facilities typically provided by relational systems, such as views, automatic query optimization, transaction management, concurrency control, dynamic schema evolution, access authorization, and triggers. Importantly, it allows large unstructured data to be stored and managed in native operating system files as though they were internal to the UniSQL/X database. Its performance at least matches that of the best RDBMSs for standard SQL queries, and it matches that of the best ODBSs for navigational access across a cluster of related objects.

Both Illustra and UniSQL/X are new systems that have not been employed heavily yet in realistic applications. Therefore, it is too early to evaluate how they scale up in reality to applications like PSEEs. In theory, however, the combination of the capabilities of RDBMSs and ODBSs may in fact result in a flexible database system that satisfies many of the PSEE requirements.

1.4.4 Summary

Our detailed investigation of different kinds of database systems revealed that no one commercially-available database system is likely to satisfy all the requirements delineated in Section 1.3. Instead of looking for a single database system that solves all their data management problems, PSEE developers should consider the strengths and weaknesses of existing database systems and pick the one(s) that better suit their specific requirements.

Relational database systems are appropriate for storing the part of PSEE application data that is naturally represented in a table format. They provide excellent querying facilities, very good performance, and flexibility as far as using a host programming language. However, RDBMSs do not provide sufficient support for maintaining general forms of data; they do not support versioning at all, and the relational views are insufficient in general. The inability to satisfy these requirements is due in large part to the shortcomings of the SQL ANSI standard, the basis for the query language of most RDBMSs. SQL3, which is a new standard, redefines

some of the notions in the SQL ANSI standard, and comes much closer to satisfying our requirements. In adopting new object-oriented concepts, SQL3, in effect, looks very much like an object-oriented database language and has little in common with the original SQL standard. This may indicate that a convergence between the relational and object-oriented database communities may be near. Whether or not the next generation of RDBMSs will adopt SQL3 remains to be seen.

Object database systems provide data model facilities that enable the definition of general data forms, data encapsulation, and versioning of objects. ODBSs offer reasonable support for evolution and a performance sufficient for medium-sized projects. ODBSs, however, do not provide good querying facilities, their performance is still not good enough for large projects, and their support for distribution is inadequate. The tight coupling between most ODBSs and a single object-oriented programming language also hinders integration of applications written in other programming languages.

Hybrid object-relational database systems combine features from both ODBSs and RDBMSs. The integration of these features goes a long way to remedy some of the shortcomings of RDBMSs. However, hybrid databases systems are still not as widely available or stable as the other two kinds.

All three kinds of databases offer some mechanisms for extending the traditional serializable transaction model in a limited fashion. However, most commercial database systems lack well-defined and consistent mechanisms for supporting cooperating sessions (see Section 1.3.2.2).

1.5 EXPERIENCE

Having evaluated existing database technology as far as satisfying the data management requirements of PSEEs, we now look at the repositories used by existing PSEEs. To attest to the insufficiency of commercial database technology, not a single PSEE we know uses a commercial database system without significant modification. Existing PSEEs either implement their own special-purpose information management component, or they extend available database technology in significant ways. Some process support systems provide only ad-hoc information management capabilities; these do not qualify as PSEEs according to the architecture we presented in Section 1.2.

PSEEs with special-purpose repositories

The developers of the majority of existing PSEEs have opted to build special-purpose information management components with "home-grown" repositories. These include EPOS, Adele, Arcadia, and Marvel. EPOS [29] has a layered architecture that includes EPOSDB, a client-server database management system that supports versioning, a structurally object-oriented data model, and long transactions. Adele [16] implements a database based on the Entity-Relationship model, but extends the model with inheritance, triggers, events, and rules. Adele also implements configuration and version management on top of the database but integrated with it.

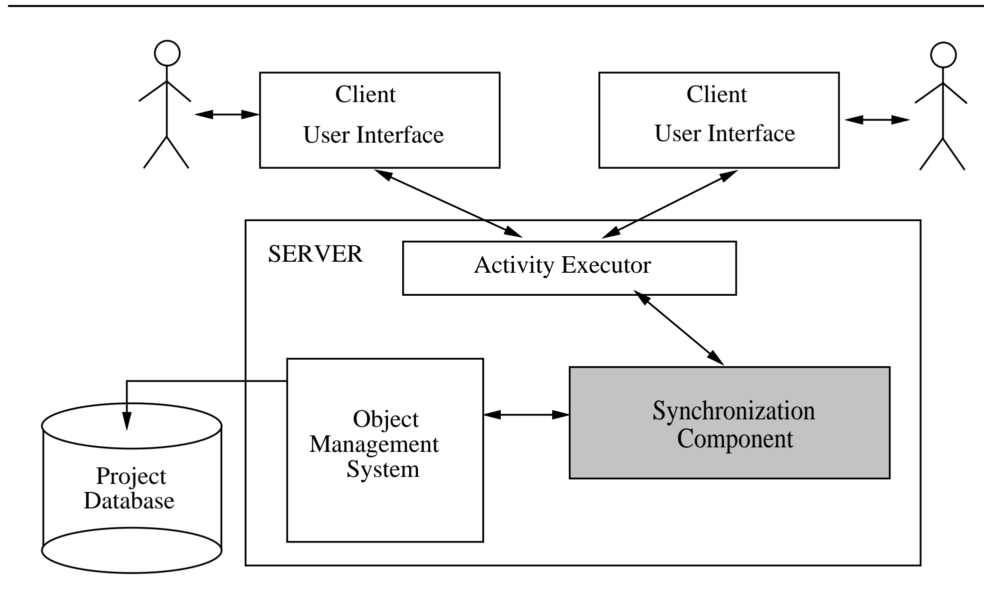

Figure 1.2 Marvel's High-Level Architecture

PSEEs that extend database technology

The PSEEs that fall in this category are built on existing database technology. However in order to obtain the required capabilities, the underlying database systems were extended in significant ways. Representatives of this category include ALF, SPADE, and Merlin. ALF [19] is built on top of PCTE/OMS, but extends it with a triggering mechanism and flexible transaction support. SPADE [10] uses O_2 but only after significant extensions, which are described later in Section 1.6.1.

In the rest of the section we present our experience in building the information management components of Marvel and Merlin. Both are representative of the present trend of building monolithic PSEEs, in which the information management component is tightly coupled with the rest of the PSEE. The discussion centers around the set of requirements discussed in Section 1.3.

1.5.1 Marvel

Marvel is a PSEE developed at Columbia University. Marvel's repository is a special-purpose object-oriented database that is implemented on top the UNIX file system. Marvel's client/server architecture [18], shown in Figure 1.2, corresponds to the generic PSEE architecture of Figure 1.1. The Project Database corresponds roughly to the repository in Figure 1.1. The Object Management System, the Synchronization Component and parts of the Activity Executor all correspond to the access management layer in Figure 1.1. Since Marvel has a single database, it does not have an integration layer. The rest of the architecture, including parts of the Activity Executor and the clients correspond to the process support and multiuser interface components in Figure 1.1.

1.5.1.1 Data composition and structure

The object-oriented database stores all application data. An object-oriented data modeling language is provided, where data types can be defined. Product data (e.g., source files, documents, etc) are maintained in UNIX files, but these files are contained in larger objects that can be instances of any of the user-defined data types. Marvel's data modeling language supports object composition and arbitrary relationships between objects.

1.5.1.2 Consistent access to data

Objects in Marvel's repository are accessed in a consistent fashion only through the database interface. All accesses to the database are controlled by a centralized server that can service concurrent requests by multiple clients. Each developer interacts with a client process in order to complete an assignment. Executing a user command (corresponds to firing a rule in Marvel), involves accessing the database for two purposes:

1. reading the values of attributes of objects, to evaluate the condition of the rule corresponding to the command;
2. to update the values of attributes of objects, based on the result of the activity.

Each access translates into a set of database operations. Each operation is a request to access a single object in the database to either read the value of one of its attributes or change the attribute's value.

The object management system (OMS) in Marvel groups the operations comprising a user command into a transaction. Transactions are created, managed and terminated by a transaction manager (TM). Thus, from the viewpoint of the repository, the *command execution layer* (which corresponds to the process enactment layer in Figure 1.1 interacts with the repository only from within transactions. The concurrency at the user activity level translates into a set of concurrent transactions, each of which composed of a set of database operations, as in the classical database model.

A lock manager (LM) employs a locking protocol to detect interference between concurrent transactions. Each transaction must obtain either a read lock or a write lock on an object before reading or writing the object, respectively. The LM grants a lock only if it does not conflict with any other lock currently held on the same object by other transactions. Otherwise, a locking conflict results, and an interference situation is detected.

In traditional database systems, when the LM detects an interference between two transactions, it notifies the TM, which resolves the interference by applying a serializability-based concurrency control policy. What is important to note is that such a policy is usually hard-wired in the database system. This limits the application system to a policy that might not be appropriate for all situations. Instead, Marvel provides a default serializability-based policy, but also a language called CRL to modify the concurrency control policy [12, 11].

A CRL specification consists of a set of *control rules*, each of which describes a class of interference situations, with varying degrees of specificity, and prescribes actions for resolving such situations. Each control rule defines a selection criterion, some variables that are bound to either transactions or lock types, and a set of condition-action pairs that use these variables. A CRL specification, when loaded into the synchronization component of Marvel, instructs the TM to resolve the kinds of interference that match the selection criterion of one of the control rules based on the condition-action pairs of the control rules rather than the

default serializability-based concurrency control policy. Resolution may include ignoring the interference if it not serious or tolerating it in various ways.

1.5.1.3 Evolution

Marvel provides mechanisms for evolving both the schema and the objects in the database.

Distribution

Marvel provides distribution only at the client level. In other words, Marvel users can access the repository from remote locations. Marvel's repository, however, is centralized and there are no provisions for distribution of data.

1.5.1.4 Integration and Interoperability

Marvel provides limited support for integration and interoperability. Since product data is maintained in files in Marvel's repository, it is readily available for external access by file-based tools. Marvel provides a programming interface called SEL [43] for defining the interface between Marvel and external tools. Marvel's repository, however, is closed and provides no mechanisms for interoperating with other database systems.

1.5.2 Merlin

The Merlin project was started at the University of Dortmund and is now continuing at the University of Paderborn. The project, in collaboration with local industry, has produced a prototype PSEE also called Merlin. Merlin uses a PROLOG-like description to describe and enact software processes. Users of Merlin are either software developers or managers involved in a project. Merlin maintains a personal agenda and an up-to-date project status information, collectively called the Working Context, for each of its users. Merlin also supports process engineers in defining and validating process models based on a graphical OMT-like description that is later on mapped to PROLOG [49].

Merlin's architecture is based on a distribution model that assumes several Working Contexts, each associated with one Process Engine. Figure 1.3 gives an overview of one such pair. A Working Context and a Process Engine communicate via the Message Dispatcher facilities based on TCP/IP. The subsystem Process Engine Control transmits working context updates to the Working Context based on the Process Engine's computation (see below). Subsystem Working Context Control is in charge of initiating the corresponding screen display and transmitting the updated status information, which is a result of user interactions and corresponding tool invocations, back to the Process Engine.

The various parts of Merlin's architecture correspond to the generic architecture we presented earlier in Figure 1.1. The Process Engine and parts of the Working Context in Merlin's architecture correspond to the process support component in Figure 1.1. The distributed database corresponds to the information management component in Figure 1.1. Parts of the Working Context correspond to the multiuser interface component.

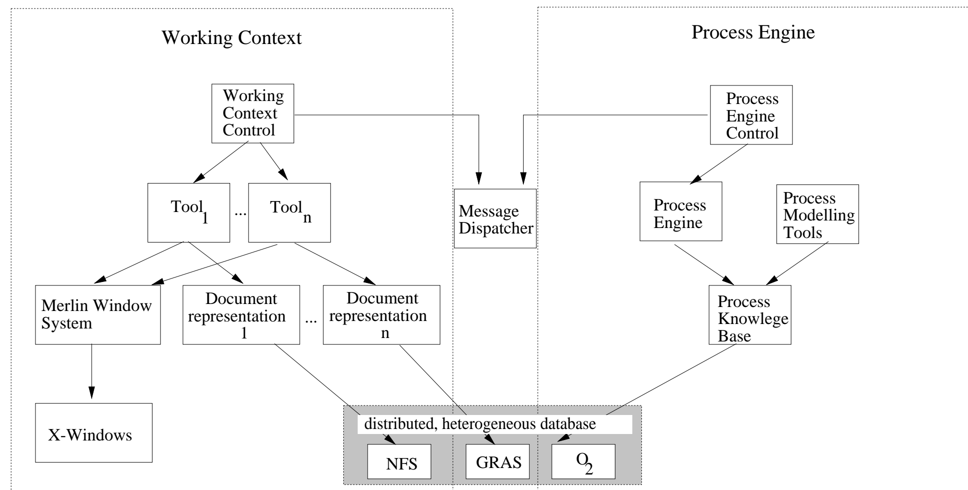

Figure 1.3 Merlin's High-Level Architecture

1.5.2.1 Data Composition and Structure

Merlin stores all process and organizational data in the ODBS O_2, and thus the discussion in Section 1.4 concerning O_2 applies here with some exceptions. Merlin currently uses O_2 only as one part of the repository. As Figure 1.3 shows, the other systems, namely GRAS and the UNIX file system, are used for product data only. Merlin originally used GRAS [57] for storing process data as well, but this had to be changed for reasons given below. Future versions of Merlin will not use GRAS anymore because we have decided to use a single database for storing both the process and product data. The reason for this is the tight coupling between process data and product data, especially when it comes to version and configuration management (cf. the GOODSTEP project described below and [63]). Instead, Merlin's extensions will be based on extending the results of the GOODSTEP project, which will provide a dedicated software engineering data base system based on an extended O_2. Members of the Merlin team are active participants in the GOODSTEP project. The UNIX file system will continue to be used to support UNIX-based tools that are integrated into the Merlin PSEE.

1.5.2.2 Consistent Access to Data

Similar to Marvel, Merlin synchronizes concurrent sessions by implementing a default synchronization policy that can be overridden by a special purpose language (basically PROLOG-like facts). Although Merlin differs from Marvel in what it provides as default synchronization policies and the mechanisms for to overriding them [65], most of the differences are not relevant to the focus of this paper. The important differences are that Merlin's policy has been implemented in PROLOG on top of O_2, and thus the Merlin Lock Manager exploits the ACID properties of transactions provided by O_2.

Basically, process data is given on the level of a single PROLOG-like fact which defines e.g. status information for some documents. This information is locked in an ACID fashion. The Merlin Lock Manager and Transaction Manager, which both belong to the architectural

component Process Knowledge Base (see Figure 1.3), implement the more flexible synchronization policies applied in Merlin. Of course, such an implementation could also be done in a DML like offered by any ODBS that is computationally complete and that offers sophisticated structuring facilities like classes. The advantage of doing it in PROLOG is that the PROLOG backtracking feature was heavily exploited to get a comprehensive and relatively simple implementation, including transaction rollbacks. The main requirement on the underlying data base system is however that it provides object-level locking (which GRAS as many other systems does not provide) to lock fine-grained data as e.g. PROLOG-like facts separately.

1.5.2.3 Evolution

Evolution in Merlin is supported through features provided by PROLOG as an interpretive language. No validation facility exists (yet) to check upfront consequences of changes before they are introduced to the running system.

1.5.2.4 Distribution

Merlin stores process data in a central repository. Access to this data is managed by the Merlin Lock and Transaction Manager as sketched above. Process engines, which compute individual working contexts, can run in a distributed client/server fashion. Future versions of Merlin will extend version and configuration management significantly. The required tight coupling between process and product data will be based on the results of the GOODSTEP project, and in particular the versioning and distribution facilities provided by O_2 (cf. Section 1.6.1).

1.6 Future Trends

After describing the present trend in building information management components for PSEEs, we now present two projects that, in many ways, represent two future trends. The first is the experience of the GOODSTEP project in extending a particular object-oriented database to make it suitable for PSEEs. The aim is to build a complete solution for the IM problem in PSEEs. The second is Provence, which takes an “open repository” approach. The paradigm followed in Provence is that there will never be a complete solution and thus the IM component must be able to interoperate with current and future storage systems. These two trends are not necessarily mutually exclusive. One could imagine a hybrid approach in which the architecture is open and includes an extended database system.

1.6.1 GOODSTEP

GOODSTEP is a European research consortium that has implemented a framework for construction of customized PSEEs. The framework is based on an extended version of O_2. The extensions were based on requirements that evolved during the initial design of the Merlin and SPADE PSEEs. The extended O_2 comprises a process modeling and enactment environment (SPADE [9]) and a generator for syntax-directed tools (see GENESIS [34]).

1.6.1.1 Data composition and structure

The process environment stores process models that are extended Petri nets in O_2. Process enactment states are stored in O_2 as well as markings of the Petri nets. Generated tools store their documents (product data) as abstract syntax graphs (ASGs) in O_2 [35]. Thus, the repository stores all application data.

1.6.1.2 Consistent access to data

Product data, stored in O_2 in the form of ASGs, are accessed via the generated tools. A tool command, such as creation of a data flow in a structured analysis tool or expansion of an export operation place holder in a modular design tool, are executed as conventional O_2 transactions that have ACID properties. This has several advantages:

- nodes of the underlying syntax-graph are only locked during the short period of time that is required to execute a command,
- only those nodes that are really read or written by the command are locked.
- only the last incomplete command is lost (wasted effort) in the case of a hardware or software failure.
- the effect of a command execution is immediately visible after commit to all developers working concurrently.

This setup enables any degree of cooperation to be defined in the process model. Atomicity and the ability to roll back sessions are handled by the process model. In particular, the process model must be able to specify checkpoints in documents to which it can return to undo a sequence of tool commands. Synchronization is handled in the process model as well. The process model must specify how to isolate concurrent developers so that their changes do not interfere. This is done through document versioning. The concurrent activities of two developers that should be isolated are assigned two different temporary variants, which are derived from a common checkpoint. When the concurrent activities are completed, the developers must manually merge the two variants, if necessary.

The extended O_2 system provides general document versioning facilities. In particular, it implements a lazy object duplication strategy, under which two versions of a composite object can share the same version of a component subobject. The shared subobject is split up into two versions only when it is modified. This ensures correct version semantics for the composite object while reducing space requirements significantly.

If not handled properly, lazy duplication may interfere with the concurrency control policy. If one transaction modifies a shared object in version `V1` that some concurrent transaction has accessed in some other version `V2`, a read-write concurrency control conflict arises. This conflict is not serious and should not cause one of the transactions to be blocked or to rollback. Instead, the shared object is split into two versions and the two concurrent transactions continue their executions. This, of course, requires that the concurrency control policy be aware of the versioning strategy. Thus, the lazy duplication strategy for version control must be implemented inside the IM together with the basic concurrency control. It is not possible to implement this strategy on top of an IM that does not support versions of composite objects.

1.6.1.3 Distribution

Distribution in O_2 is based on a client/server architecture. Execution instances of a GOODSTEP PSEE that run on different workstations can access and modify shared information. The server does not perform any significant computation, thus avoiding a bottleneck situation. Instead, the server only manages secondary storage and implements traditional (ACID) concurrency control. Both facilities have to be implemented in the server because it is the only entity that knows which objects are accessed by clients. Methods defined in the database schema are executed on clients rather than on the server, as in Marvel. Objects are transferred via remote procedure calls (RPCs) from the server to the clients and vice versa, with the overhead of RPC communication.

GOODSTEP PSEEs can define and manage fairly fine-grained objects. These objects are stored as nodes of ASGs or as places, transitions and tokens of petri nets. To read a complete document or process model, a potentially large number of objects may need to be transferred between the server and the client. A medium-sized document, for instance, may consist of 1,000 to 10,000 nodes. An RPC requires between 2 and 4 milliseconds. If each RPC transfers only a single object, the performance of the PSEE becomes unacceptable (up to 40 seconds for loading a medium-size document). To enhance performance of object transfer, the granularity for distribution in O_2 are copies of secondary storage pages rather than single objects. The clustering scheme in O_2 stores between 50 and 200 related small objects in a single page [20]. This reduces network communication overhead significantly.

Unfortunately, this granularity for distribution interferes with the concurrency control mechanism of the database server. Originally the granules that were locked in O_2 were pages as well. The server locked a page before transferring it to a client. This resulted in a large number concurrency control conflict even when concurrent transactions accessed disjoint sets of objects. To remedy the situation, the page-level concurrency control scheme of O_2 was extended by the GOODSTEP project to a two-level scheme. Basically, if a concurrency control conflict arises when accessing a page, the scheme switches to object-level locking.

1.6.2 Provence

The Provence [53] project follows a different approach than the GOODSTEP project. Rather than developing a single database that meets all the requirements and that controls all application data, Provence assumes an open repository that separates between information needed for the enactment of a software process model and information needed for the actual execution (or performance) of a process [15, 13]. The premise is that the actual execution of a project's processes takes place within a working environment that comprises an integrated set of tools and storage system utilities. Organizations may store their product data in files, for example, and access this data via tools like compilers, editors, debuggers, formatters, and viewers. Provence aims to provide process-centered assistance to these organizations without radically changing their working environments.

To achieve this, the Provence architecture, shown in Figure 1.4, separates the process model enactment domain from the process execution domain both logically and physically. A process support component (PS) controls the process model enactment domain while an event monitor (EM) monitors the process execution domain. A translator component acts as a liaison between the PS and the EM. A *virtual repository* contains the data generated and maintained in both domains. The virtual repository includes an actual repository under the control of the PS,

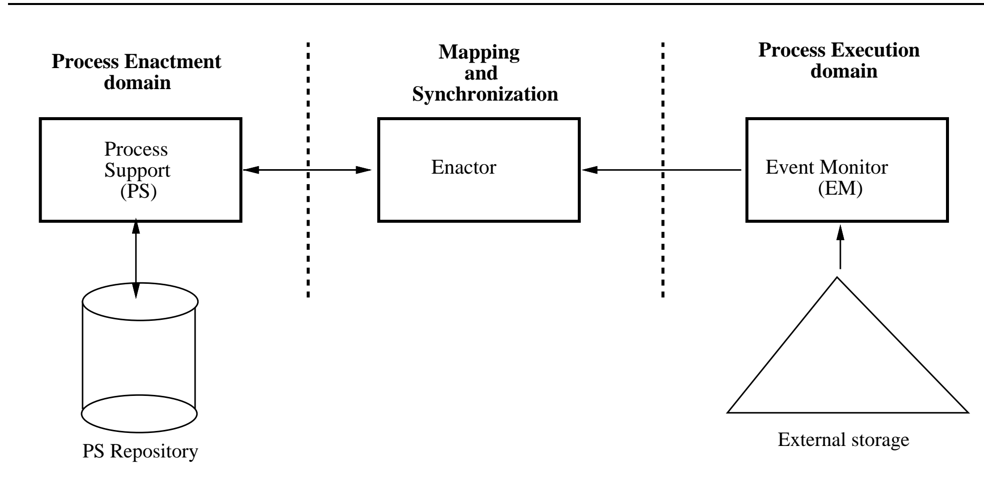

Figure 1.4 Open Repository in Provence

referred to as the PS repository hereafter, and a collection of external objects controlled by external storage systems, such as the underlying file system. In the current implementation of Provence, Marvel [14] is used as the PS, and Marvel's object database is used as the PS repository.

The PS enacts a model of the project's processes. This model has three parts: (1) a set of classes that describe the project's application data; (2) a specification of process transitions, which correspond to process steps; and (3) process enactment rules, which prescribe how and/or when process transitions should/can take place. During the enactment of the process model, the PS maintains enactment information in the PS repository.

Part of the application data (mostly the product data) is stored outside the PS repository and thus is not under the control of the PS. For each external object that is part of this external application data, the PS creates a mirror object in PS repository. A mirror object does not duplicate any of the contents of the corresponding external object, but stores enactment-relevant information about the object, such as when it was modified last. The translator component, called the *Enactor*, is responsible for synchronizing the state of the external objects and the state of the mirror objects in the PS repository.

The Enactor performs two tasks. First, it maps the subset of the enactment transitions involving mirror objects into expected sequences of executable events that occur on the external objects. Second, it collects information about the actual occurrence of such event sequences from the EM, maps them into the corresponding enactment transitions, and notifies the PS. The Enactor thus translates a sequence of state changes in the actual execution of the processes into one state transition in the process model enactment.

1.6.2.1 Example

Consider a network maintenance process: when a sensor detects a failure, it logs the error in a special file called `alarm`; when a human operator detects the entry in `alarm`, s/he retrieves the error log and forwards it to the failure resolution team. The file `alarm` is stored outside the PS repository, but a mirror object is stored in the PS repository. The mirror object has four

attributes: path, modified, accessed, and timestamp. Path stores the pathname of the external file. The value of modified reflects whether or not the file has been modified since the last synchronization point. Similarly, the value of accessed reflects whether or not the file has been accessed (read) since the last synchronization point. Timestamp stores the timestamp of the last synchronization point involving this object.

The intention is to have the insertion of a failure log into the file alarm somehow trigger an enactment transition to change the value of the attribute modified of the corresponding mirror object to reflect that. More specifically, this enactment transition corresponds to a sequence of three external events: opening the file alarm, writing an entry into it, and closing it.

All mirror objects in the PS repository are identified as such. For each one of these objects, the Enactor generates a set of executable events that the EM should monitor on the actual external object corresponding to the mirror object. The kinds of event to monitor depend on the type of mirror object (it can be one of five types in Provence: Monitored File, Monitored Directory, Monitored Tool, Temporal Event, Announced Event). Thus, for the mirror object of alarm, the Enactor asks the EM to monitor the occurrence of the three events listed above (opening, writing and closing the file) on the actual file.

When the EM detects the occurrence of the three external events, it notifies the Enactor. The Enactor maps the three events to the corresponding enactment transition and notifies the PS. The PS performs the transition, bringing the enactment state in sync with the execution state. In addition, the PS may perform other enactment transitions.

Of course, the actual execution of the processes may deviate from the process model; the Enactor and the Process Server can detect such deviations and handle the exception. Exception handling actions range from sending notifications about the deviations to rolling back the actual execution steps if possible. Discussing this important matter in more detail requires a separate paper.

There are two main advantages to the approach followed by Provence: minimization of intrusion and facilitation of process model evolution. The architecture minimizes intrusion because project personnel have the choice of executing the project's processes in their current working environment, using the same set of tools and the same operating systems facilities that they would have used without Provence. In addition, introducing new file-based tools does not require any integration with the PSEE. The human users do not have to interact with the Process Server except indirectly (e.g., through notifications). Monitoring of the execution of the processes is done at the level of operating system events, and thus does not affect the working environment and does not hinder the evolution of that working environment in any way. In other process-centered systems, such as Marvel, the users are directly involved in the enactment of the process model, and their interactions as well as their activities are directly constrained by the model enactment; the users, in fact, access external tools, the file system, external databases, etc., only via the process-centered system.

The second benefit is that the architecture allows for evolving the process model without affecting process execution. Thus, process model entities and enactment rules can be added, modified, or deleted while the actual processes are being executed. Since the mapping and synchronization between the model enactment domain and the process execution domain is handled by a separate component, it is possible to handle the necessary changes that result from the evolution of the process model without affecting the process execution. If the evolution results in the need to monitor additional events, the Enactor can convey the appropriate specifications to the Event Monitor, which will start monitoring these events from that point

on. A mechanism must be devised so as to avoid missing significant execution events that are in progress when the evolution occurs.

1.6.3 Summary

GOODSTEP and Provence represent two future trends in PSEE architecture, as far as information management is concerned. GOODSTEP promises a complete solution to the information management problem by extending an ODBS with facilities that satisfy all the requirements discussed in Section 1.3. Provence espouses the premise that any solution is incomplete due to the evolving nature of software tools and environments, and thus the most viable approach is to provide an open system that is easily integrated with existing environment but that can readily incorporate changes. Each approach has advantages and disadvantages; future experience will shed more light on which approach is the more viable trend.

1.7 Conclusion

The paper explored the information management problem in PSEEs. It characterized the forms of data maintained by PSEEs and the access to this data, and it delineated a set of data management requirements based on these characteristics. It evaluated existing database technology from the viewpoint of these requirements. Finally, the paper described past and present trends in managing information in PSEEs, using our own experience as a representative of these trends.

The past trend in information management in PSEEs illustrates clearly the insufficiency of current database technology — not a single existing PSEE uses a commercial database system without extensions for its repository. Instead, special-purpose information management systems have been developed to meet the requirements of specific PSEEs. The IM component in these PSEEs is typically tightly coupled with the rest of the PSEE, resulting in a monolithic system that is hard to extend.

The present trend shows that many PSEEs are experimenting with extended object database systems (ODBSs). ODBSs already satisfy more of the information management requirements than RDBMSs, but they are less mature than RDBMSs. However, work is in progress to make ODBSs more mature and suitable for PSEEs. Areas for improvement include standard query language, extended transaction management for cooperation and coordination, integrated version and configuration management, transparent distribution of data, and interoperability with applications written in a variety of programming languages. Several PSEEs are being developed on top of ODBSs that are being enhanced specifically to satisfy the requirements of PSEEs.

There are two future trends that have taken shape. The first is towards providing complete solutions to software problems, including information management in PSEEs. The GOODSTEP project is an active promoter of this trend. The second trend is towards componentization, interoperability and openness. The Provence project represents an example of that; the Adele project is also trying a similar approach. The advantages and disadvantages of each approach, some of which are apparent today, will become clearer when they are tried in practice. The two trends are also not necessarily in opposition of one another. In fact, a hybrid approach based on an open architecture that includes an extended ODBS is indeed possible. Whether or not such a hybrid approach materializes in the near future remains to be seen.

ACKNOWLEDGMENTS

We are indebted to the members of the Merlin and Marvel projects for some of the information in this paper. Gail Kaiser was the principal investigator of the Marvel project at Columbia University; Michael Sokolsky, Israel Ben-Shaul and George Heineman were instrumental in developing the object management component and the database of Marvel. Stefan Wolf and Jim Welsh contributed significantly to the development of the information management component of Merlin. We thank Joëlle Madec, Claude Delobel and Fernando Velez for the intense discussions about the relationship between distribution, concurrency control and version management. We also thank Donald Caldwell and Rick Greer at Bell Labs for their help in clarifying the functions and features of commercial relational database systems. Finally, we thank the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

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

[6] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems*, 6(4):557–575, 1982.

[7] S. Bandinelli, L. Baresi, A. Fuggetta, and L. Lavazza. Requirements and Early Experiences in the Implementation of the SPADE Repository using Object-Oriented Technology. In S. Nishio and A. Yonezawa, editors, *Proc. of the First JSSST International Symposium Kanazawa, Japan*, volume 742 of *Lecture Notes in Computer Science*, pages 511–528. Springer, 1993.

[8] S. Bandinelli, L. Baresi, A. Fuggetta, and L. Lavazza. Experiences in the Implementation of a Process-Centered Software Engineering Environment using Object-Oriented Technology. Technical Report RT94026, Centro CEFRIEL, Politecnico di Milano, October 1994.

[9] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Luigi Lavazza. Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, 1993.

[10] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Luigi Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. In *[40]*, pages 223–248. 1994.

[11] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University Department of Computer Science., February 1992. Technical Report CUCS-001-92.

[12] Naser S. Barghouti. Supporting Cooperation in the Marvel Process-Centered SDE. In *Proc. of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 21–31, Tyson's Corner, VA, December 1992.

[13] Naser S. Barghouti. Separating process model enactment and execution in provence. In *Proc. of the 9th International Software Process Workshop*, Airlie, VA, October 1994. IEEE Computer Society Press.

[14] Naser S. Barghouti and Gail E. Kaiser. Scaling Up Rule-Based Development Environments. In *Proc. of 3rd European Software Engineering Conference, ESEC '91*, pages 380–395, Milan Italy,

October 1991. Springer-Verlag. Published as *Lecture Notes in Computer Science* no. 550.

[15] Naser S. Barghouti and Balachander Krishnamurthy. An open environment for process modeling and enactment. In *Proc. of the 8th International Software Process Workshop*, Schloss Dagstuhl GERMANY, March 1993. IEEE Computer Society Press.

[16] Noureddine Belkhatir, Jacky Estublier, and Welcelio Melo. ADELE-TEMPO:An Environment to Support Process Modelling and Enaction. In *[40]*, pages 187–221. 1994.

[17] Israel Z. Ben-Shaul and Gail E. Kaiser. Process Evolution in the Marvel Environment. In Wilhelm Schafer, editor, *8th International Software Process Workshop: State of the Practice in Process Technology*, pages 104–106, Wadern, Germany, March 1993. Position paper.

[18] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. In *ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 149–158, Washington D.C., December 1992.

[19] Benali, K. *et al.* Presentation of the alf project. In *Ninth International Conference on System Development Environments and Factories*, Berlin Germany, May 1989.

[20] V. Benzaken, C. Delobel, and G. Harrus. Clustering Strategies in O_2: An Overview. In *[5]*, pages 385–410. 1992.

[21] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading MA, 1987.

[22] Philip Bernstein. Database System Support for Software Engineering – An Extended Abstract. In *Ninth International Conference on Software Engineering*, pages 166–178, Monterey CA, March 1987. IEEE Computer Society Press.

[23] E. Bertino. A View Mechanism for Object-Oriented Databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology — EDBT'92, 3rd Int. Conf on Extending Database Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 136–151. Springer, 1992.

[24] T. 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.

[25] J. Brunsmann. Versions- und Konfigurations-Verwaltung in syntax-gesteuerten Software-Entwicklungswerkzeugen. Master's thesis, University of Dortmund, Dept. of Computer Science, Software Technology, 1994. Forthcoming.

[26] R. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufman, 1993.

[27] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[28] C. Collet, T. Coupaye, and T. Svensen. NAOS Efficient and modular reactive capabilities in an Object-Oriented Database System. In *Proc. of the 20th Int. Conf. on Very Large Databases, Santiago, Chile*, 1994. To appear.

[29] Conradi, Reidar *et al.* EPOS: Object-Oriented Cooperative Process Modelling. In *[40]*, pages 33–70. 1994.

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

[31] 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.

[32] 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.

[33] ECMA. Introducing PCTE+. Technical Report ECMA/TC33/89/48, Independent European Programme Group – Technical Area 13, 1989.

[34] W. Emmerich. *Tool Construction for Process-Centred Environments based on Object Database Systems*. PhD thesis, University of Paderborn, 1995. Forthcoming.

[35] W. Emmerich, P. Kroha, and W. Schäfer. Object-oriented Database Management Systems for Construction of CASE Environments. In V. Mařik, J. Lažanksý, and R. R. Wagner, editors, *Database and Expert Systems Applications — Proc. of the 4th Int. Conf. DEXA '93, Prague, Czech Republic*, volume 720 of *Lecture Notes in Computer Science*, pages 631–642. Springer, 1993.

[36] Wolfgang Emmerich, Wilhelm Schäfer, and Jim Welsh. Databases for Software Engineering – The Gaol has not yet been attained. In *Fourth European Conference on Software Engineering*, pages 145–162, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag. Published as *Lecture Notes in Computer Science* no. 717.

[37] Gregor Engels, Claus Lewerentz, Manfred Nagl, Wilhelm Schaefer, and Andy Schuerr. Building Integrated Software Development Environments. *ACM Transactions on Software Engineering and Methodology*, 1(2), 1992.

[38] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–632, November 1976.

[39] F. Ferrandina, T. Meyer, and R. Zicari. Implementing Lazy Database Updates for an Object Database System. In *Proceedings of 20th International Conference on Very Large Data Bases*, 1994.

[40] Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Ltd., John Wiley & Sons Inc., Taunton, England, 1994.

[41] 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.

[42] J. Giavotto, G. Rosuel, A. Devarenne, and A. Mauboussin. Design Decisions for the Incremental Adage Framework. In *Proc. of the 12th Intl. Conf. on Software Engineering, Cannes, France*, pages 86–95, 1990.

[43] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In *1st International Conference on the Software Process*, pages 218–227, Redondo Beach CA, October 1991.

[44] GOODSTEP Team. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In K. Ohmaki, editor, *Proc. of the Asia-Pacific Software Engineering Conference, Tokyo, Japan*, pages 410–420. IEEE Computer Society Press, 1994.

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

[46] G. Gottlob, P. Paolini, and R. Zicari. Properties and Update Semantics of Consistent Views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

[47] S. Heiler and S. B. Zdonik. Object Views: Extending the Vision. In *Proc. of the 6th Int. Conf. on Data Engineering, Los Angeles, CA*, pages 86–93. IEEE Computer Society Press, 1990.

[48] S. E. Hudson and R. King. The Cactis Project: Database Support for Software Environments. *IEEE Transactions on Software Engineering*, 14(6):709–719, 1988.

[49] G. Junkermann. A Prolog-based Semantics of a Dedicated Process Design Language. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, 1995. To appear.

[50] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In *[40]*, pages 103–129. 1994.

[51] W. Kim. Object-Oriented Database Systems: Promises, Reality, and Future. In *Proceedings of 19th International Conference on Very Large Data Bases*, 1993.

[52] 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.

[53] B. Krishnamurthy and N. Barghouti. Provence: A Process Visualization and Enactment Environment. In *Proc. of the Fourth European Conference on Software Engineering*, pages 151–160, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag. Published as *Lecture Notes in Computer Science* no. 717.

[54] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[55] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):51–63, 1991.

[56] C. Lécluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proceedings of the 1989 ACM SIGMOD Int. Conf. on the Management of Data, Portland, OR*, pages 424–433, 1988.

[57] C. Lewerentz and A. Schürr. GRAS – A Management System for Graph-like Documents. In Beeri, Schmidt, and Dayal, editors, *Proc. of the 3rd Conference on Data and Knowledge Bases*, pages 19–31. Morgan Kaufman, 1988.

[58] 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.

[59] J. Madec. Version Management in O_2. Technical report, O_2-Technology, 1993.

[60] Jim Melton and Alan Simon. *Understanding The New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[61] M. Penedo, E. Ploedereder, and I. Thomas. Object Management Issues for Software Engineering Environments. In P. Henderson, editor, *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 226–234, Boston MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[62] *ACM SIGMOD Workshop on Software CAD Databases*, Napa CA, February 1989. ACM Press.

[63] S. Sachweh and W. Schäfer. Version management for tightly integrated software engineering environments. In *Proc. of the 7th Int. Conf. on Software Engineering Environments, Leiden, The Netherlands*. IEEE Computer Society Press, 1995.

[64] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In *Proc. of the 4th Int. Conf. on Extending Database Technology, Cambridge, UK*, Lecture Notes in Computer Science. Springer, 1994.

[65] W. Schäfer and S. Wolf. Cooperation Patterns for Process-Centred Software Development Environments. In *Proceedings of the 7th International Conference on Software Engineeing and Knowledge Engineering*, 1995. To appear.

[66] H.-J. Schek and G. Weikum. DASDBS – Concepts and architecture of a novel database system (in German). *Informatik Forschung und Entwicklung*, 2(3):105–121, 1987.

[67] 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 2nd Int. Conf., DOOD '91, Munich, Germany*, volume 566 of *Lecture Notes in Computer Science*, pages 189–207. Springer, 1991.

[68] A. H. Skarra and S. B. Zdonik. Type Evolution in an Object-Oriented Database. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, Cambridge, MA, 1987.

[69] Special Working Group on Ada Programming Support Environments. International Requirements and Design Criteria for the Portable Common Interface Set. PCIS Technical Report, PCIS, July 1992.

[70] M. Stonebraker. The Miro DBMS. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

[71] Michael Stonebraker, Eric . N. Hanson, and Spyros Potamianos. The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[72] Walter F. Tichy, editor. *Configuration Management*, volume 2 of *Trends in Software*. John Wiley and Sons Inc., 1994. B. Krishnamurthy (series ed.).

[73] M. Ubell. The Montage Extensible DataBlade^{TM} Architecture. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994.

[74] J. Walpole, G. Blair, J. Malik, and J. Nicol. A Unifying Model for Consistent Distributed Software Development Environments. In P. Henderson, editor, *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 183–190, Boston MA, November 1988. ACM Press. Special issue of *SIGPLAN Notices*, 24(2), February 1989.

[75] 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.

[76] ANSI X3.135-1992. *American National Standard for Information Systems – Database Language – SQL*. American National Standards Institute, 1992.