Chapter 5
Architectural Views and Alternatives

Editor: Wilhelm Schaefer
Contributors: Alfonso Fuggetta, Claude Godart, Jens Jahnke

5.1 Basic Components

A broad overview of the key components of a PSEE has been provided in Section 1.6 (Process-sensitive Software Engineering Environments on page 10). In this chapter we will refine the definition of each of these components and discuss different architectural alternatives. We start in Section 5.1.1 with the definition of a general reference model for architectures of PSEEs to enable us to reason about alternative approaches. The components of the proposed reference architecture are motivated by the requirements for basic services in a PSEE.

In Sections 5.1.2 — 5.1.7 we provide a general description of each service and discuss different alternatives for their implementation in existing Promoter systems and others.

In Section 5.2 we discuss particular requirements and architectural alternatives for distributed environments and finally, a concrete example in Section 5.3 describes in more detail the architecture and implementation of the distributed PSEE ‘Merlin’ [Junk94].

5.1.1 A Reference Model for Architectures in PSEEs

A general reference architecture for PSEEs can be derived by identifying a number of required basic services. To begin with, each PSEE needs a dialog management to inform users about process information and enable them to perform activities. The task of process management is to execute a particular process model and to coordinate concurrent activities of multiple users. A personal workspace is maintained for each user in his/her particular role. It includes all software objects that have to be accessed by this particular user/role combination. Software objects and corresponding relations are persistently stored and have to be efficiently accessible through the PSEE’s repository management.

As represented in Figure 5.1, these basic services are organised as four architectural abstraction layers built one on top of another. There are various alternatives as to how these services layers may be implemented in components of a particular PSEE. For example in one PSEE (e.g. Merlin [Junk94]), workspaces are managed by the process...
Architectural Views and Alternatives

Engine only while in others (like COO and Adele) this is part of a sophisticated repository component..

We can identify a further basic service referred to as communication management which is needed to exchange messages (notifications and requests) between the different components of a PSEE and the integrated software development tools. We shall now examine each component in some detail.

5.1.2 Dialog Management

The dialog management layer encapsulates the user interface of a PSEE. Typically, users of a process centred environment can interact with different roles. Generally, we can distinguish between the roles of software developers, project managers and process engineers. There are different interaction patterns for each of these roles.
The knowledge about a software project that has to be maintained by a PSEE typically tends to be somewhat extensive. For a single member of the project in the role of a developer only a small part of this knowledge is relevant. He/she therefore requires the assistance of the PSEE to display all the software artifacts that he/she needs to access or is responsible for. This information can be seen as an external representation of a personal workspace (see Section 5.1.4). Furthermore, depending on the current state of the project, the environment should provide developers with personal agendas which include information about all possible activities that can be performed. In order to provide the user with a clear overview of more complex workspaces, filters may be applied as a means of abstraction. Customised filter settings enable the user to hide less important parts of a workspace representation.

When an activity is performed whose result influences its own or other workspaces (e.g. document states have changed or new documents have been created), these workspaces have to be updated. Such updates should not automatically trigger a refresh of the corresponding workspace representations. This is in order to avoid more or less continuous refreshes due to activities performed by other users, which would be highly ‘user-unfriendly’. One solution, applied in Merlin, is to inform a user about a pending update of his/her workspace representation by the indication of a special update flag. The user can then decide when the changes are to be propagated in his/her workspace. Such changes could for example be that a number of new documents are added to the workspace or a menu is extended by a further item indicating activities which can be performed on the corresponding objects.

When executing a process, software developers are directly influenced by the conflicts which occur (see Chapter 6). In case of automatic synchronisation, the dialog interface must at least report on these conflicts and their consequences. In the case of interactive conflict resolution, the developer contributes directly to synchronisation.

Project managers require tools to instantiate projects, to (re-)assign project members with roles and responsibilities and to specify further constraints like milestones and deadlines. Not only project managers, but other roles such as developers should also be able to get a complete overview of the project including information that is not automatically displayed within the representation of a single workspace. Such information includes for example the project history, defined milestones, responsibilities and roles of other developers. Moreover project managers and developers should be able to query the environment about the impact of a particular activity.

Finally, the process engineer requires tools to create new process models and also to change existing process models “on the fly” using an adequate PML (see Section 3.2). These tools should provide means to analyse, validate and improve process models. For example this may be achieved by the definition of metrics for software process models that can be derived from experiences with instantiated projects.
5.1.3 Process Management

The process management component of a PSEE coordinates the different activities of multiple developers involved in a software project. For each user involved in a software development project it computes his/her specific workspace (see Section 5.1.4) reflecting the current state of the project. It offers activities on documents according to preconditions, e.g. the roles which could perform the activity, the responsibilities of a person, the corresponding access rights to the appropriate documents, or the dependency with other documents.

The process management service should also be able to explain current and previous project states on request. For instance the environment should be able to answer user inquiries such as “Why should I code module m1_c?”, “Who else is involved in the project?”, “Who changed the specification for module m1_c?”, or “What are the time constraints for coding m1_c?”, i.e. questions which could be asked by using the interaction facilities of Dialog Management (see Section 5.1.2).

This service layer can include one central or many distributed Process Engines (see Section 5.2) which execute a formal description of the software development process which we from now on refer to as the Software Process Program (SPP). An important requirement is that SPPs have to be capable of modification during process execution because such a process cannot usually be fully determined in advance. An SPP can be divided in three layers according to their stability during process execution, namely the Cooperation Model and the Process- and Project-Layers.

The Cooperation Model is the most stable part of the SPP. It has to be changed only when the user interface paradigm (see Section 5.1) is changed or new features like configuration management are added. It provides a predefined set of rules which constitute the basis for process execution, i.e. it acts as an inference machine for the description of the other two layers. The rules in the Cooperation Model again can be separated according to their concerns, e.g. transaction management (see predefined strategies in Section 6.3), configuration management, state transitions, and computation of workspaces.

**Figure 5.2** Software process program layers
5.1 Basic Components

The **Process-Layer** represents the type view of the actual software development process. The description in this layer includes for example document types, tool types, possible states and state transitions. Changes to the Process-Layer may occur during execution of a concrete software development project, because parts of the process themselves depend on decisions made during the course of the process e.g. introduction of a new test strategy, new test cycles, or the introduction of a new experimental development path.

Finally, the **Project-Layer** includes all information that is needed for process instantiation. The main information that is provided in this layer consists of the names, roles and responsibilities of people participating in a project and the types and names of documents to be produced. Changes to the Project-Layer may occur frequently because of unforeseen events such as sickness of staff, lack of skills, and break-down of machines.

In existing approaches, the different layers of an SPP may be specified using the same or different languages. For example in SPADE one single PML (SLANG) is used to specify the Process-Layer and Cooperation Model whilst Merlin uses ESCAPE [Junk95] for the specification of the Process-Layer and PROLOG to define the cooperation model.

The requirement for changes to an SPP even during the course of process execution (“changes on the fly”) requires an interpretative approach as the basis for process execution which renders a compiler unusable.

5.1.4 Workspace Management

The workspace for a particular user in a particular role at a certain state of the project is defined to include only those software objects and relations which are needed in order that his or her task might proceed.

The two basic motivations underlying the innovation of the workspace management layer are **abstraction** and **isolation**. Workspaces enable users to concentrate on their specific task in the project by abstracting away irrelevant information about other parts of the project. Furthermore they contribute to the avoidance of unintended or unauthorised manipulation of software objects, i.e, to preserve their integrity.

However strict isolation is not practicable for software processes which are cooperative by nature. Indeed, developers must both be able to work in isolation when they want, and to share objects and to synchronise changes on shared objects in different workspaces, generally on their initiative. The cooperation of multiple developers has to be coordinated by an advanced transaction mechanism (see Chapter 6).

There are different alternatives for the implementation of the workspace management layer. In many approaches (e.g. COO, ADELE) workspaces are implemented using a Base/SubBase architecture which enables nested workspaces. The workspace manager then provides capabilities to transfer object copies in a SubBase where developers can work in isolation. This basic capability is extended to allow sharing of object modifica-
tions. This is specified by the definition of an advanced transaction mechanism (see Section 6.2.8.1).

An alternative implementation (e.g. in Merlin) is to have a common representation for shared software objects. In this approach workspaces represent logically different views on physically identical objects, corresponding to view mechanisms in database systems. Merlin provides an advanced transaction mechanism (see Section 6.4.2) and version and configuration management to meet the requirement for isolation.

5.1.5 Repository Management

The Repository Management service is responsible for maintaining consistency and availability of the information needed by the other PSEE components. Typically, this information is concurrently accessed by many different PSEE users. The Repository Management has to resolve conflicting accesses and is thus a key component of the architecture of a PSEE.

Evaluations of currently available relational and object-oriented database systems show that there is no single commercial system that completely satisfies all PSEE Repository Management requirements. Relational DBMSs are a mature technology that provides excellent support for querying, distribution, and 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 databases 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 databases are tightly coupled to a single programming language, complicating integration with applications that are written in a different language.

In the following we will sketch requirements that have to be fulfilled by a suitable Repository Management to build the basis for a PSEE.

5.1.5.1 Data Composition and Structure

A software development project typically generates many different forms of data, as illustrated by the following (non-exhaustive) 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.
- **organisational 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. As an example consider the document graph in Figure 5.3 that consists of all existing documents at a specific state of a project as its nodes, connected by directed edges representing inter-document relationships.

These documents 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, Figure 5.3 shows that 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 inter-object 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).

![Figure 5.3 Data Model: Different Levels of Granularity](image)

The Repository Management 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) whilst maintaining relationship information among them.
5.1.5.2 Multi-User Support

In most projects a number of users work in parallel on different parts of the overall project activity. The Repository Management therefore has to be capable of scheduling concurrent accesses of multiple users. Conflicts (deadlocks) have to be avoided through sophisticated locking strategies or they have to be detected and resolved.

5.1.5.3 Efficiency

The state of the project instance changes whenever a new document is introduced or an existing document is deleted, when a document is declared to depend on some other document, when a document becomes complete or when it becomes incomplete due to a change in some other document on which it depends. Although this list is incomplete, it is sufficient to indicate that changes to project states occur frequently, and all cause recomputation of the workspaces of all the users who are affected by the changes. This computation must be done efficiently, since the user expects his/her workspace to be consistent with the current state of the project.

Furthermore, in order to achieve reasonable response times for syntax directed software development tools an efficient repository is required to store, retrieve and analyse fine grained information, e.g. abstract syntax trees or graphs.

As early as 1987, Bernstein argued that dedicated database systems for software engineering environments, specialised with respect to functionality and implementation, were necessary [Bern87]. He and others [Tayl88] argued that the functionality and efficiency of standard database systems (in particular relational systems) did not adequately support the construction of software engineering tools and environments.

The main reason for the inefficiency of relational systems is that a flat (normalised) internal database schema (tuples) does adequately represent high-level data structures used in software engineering environments, e.g. relations between complex nodes in abstract syntax graphs. This results in expensive transformation procedures between both representations.

Existing approaches prove that object-oriented database systems are a suitable platform for the implementation of PSEEs. For example Merlin and SPADE both use the commercial object management system O₂ [Deux91] while COO uses P-Root [Char93, Char94] which is an implementation of PCTE.

5.1.5.4 Persistency and Integrity

Like most software programs, the PSEE may need to be stopped from time to time and so it must be able to store the state of the project persistently in order to prepare a restart. Even if it is stopped accidentally e.g. by hardware or software failure, it must resume with a consistent project state, and such a failure must not result in a significant loss of project information. Thus, it is required that the Repository Management preserves integrity of any project information, i.e. it ensures that continuation of any operation is possible after failure.
5.1.5.5 Distribution

Multi-user support usually means distribution of the project activities over a number of single-user workstations. There are basically two ways to achieve this. The first is to have a single server for the repository and to allow distributed access from the user’s client workstation. The second way is to distribute transparently the project information itself over various repositories that are locally accessible from the user’s workstations.

With the first approach the single server would certainly become a performance bottleneck for the whole PSEE, hence this approach seems feasible only for small projects. It is, however, worth consideration as many projects are either small or can be split into fairly independent sub-projects that are themselves sufficiently small.

With the second approach, the Repository Management can arrange that parts of the project information are represented in local repositories of users which are responsible for these parts. The actual distribution of the project information should be transparent for the tools that use the repository. It should rather be responsibility of the Repository Management to manage physical distribution. This latter approach reduces traffic on the communication network as well as the amount of expensive remote access to common project information, allowing larger projects to be handled.

5.1.5.6 Heterogeneity

In Section 5.1.6 we proposed an open system architecture for PSEEs which supports the integration of different SDTs at various levels of granularity. In order to achieve this the underlying repository must be able to maintain different parts of the project information in heterogeneous representation. For example a module that has been implemented with a syntax directed editor is stored as an abstract syntax tree in a non-standard database system, while the corresponding error report is an ASCII file in the Unix file system.

5.1.5.7 Evolution

If there is a feature that is common to every software system, it is this: the system will undoubtedly need 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 localised. For example, a developer may check out a module, modify it, test the modified version, and check it back in again. 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 minimise check-out conflicts among developers. The most common mechanism with which a PSEE supports simultaneous development is a versioning and configuration management facility [Tich94]. The Repository Management 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 [Naka91, Barg93], delayed or lazy conversion [Ferr94], and versioning of the schema [Skar87]. 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.

5.1.5.8 Version and Configuration Management

An analysis of some of the most representative transaction models for PSEE, as described in Chapter 6, points out the importance of version management in advanced transactions. Thus, the Repository Management layer has to support the management of versions of objects and configurations of the set of objects defined in a project. In particular, it must enable its clients to derive new versions of (composite) objects, build new configurations (by the selection of object versions) and to maintain a version history. For example, the above mentioned object management system O2 which is used by Merlin and SPADE provides sophisticated versioning mechanisms.

5.1.5.9 Flexible Transaction Management

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. This kind of 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 characterised 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 to the transactional interaction between the user of a banking application, for example, and a database management system. Process knowledge can be used to relax serialisability of traditional transaction models as required by software processes.
A detailed description of non-standard transaction models in the application domain of software engineering is provided in Chapter 6. In any case, there is always a level of abstraction at which a process is a serial execution of ACID short-term transactions, so the Repository Management layer must provide ACID transactions.

5.1.5.10 Ad-Hoc Query Facilities

The Process Management needs to query the current project information in order to extract information about the states of documents upon which the process engine must base its decisions. An example of such a query could be: “Select all program modules, for which programmer Miller is responsible that are incomplete but whose specifications are complete.” Such queries are not known in advance, so Repository Management has to offer ad-hoc query facilities for use by Process Management.

5.1.6 Communication Management

Best user acceptance can be achieved if a PSEE has an open architecture which supports different levels of integration. The architecture has to allow a customisable, highly flexible and extensible environment, otherwise the environment might be regarded as counter-productive. This is because:

• obviously project support has to match the various kinds of projects,
• an environment should be adaptable to changing needs as a project proceeds,
• an environment should be partially adaptable to new state-of-the-art technologies (exchange of old tools) and
• it should be possible to add new tools when suitable tools for previously-unsupported areas become available.

An adequate technique to achieve an open environment satisfying the above mentioned requirements is to use a communication framework to integrate different parts of the environment rather than compiling all parts of the architecture into a highly integrated but inflexible monolithic environment.

An example of an available communication framework which is used in SPADE is DEC FUSE\(^1\) [DEC91]. It provides a predefined set of messages for tool integration. Each integrated tool exports a set of services that can be invoked by other tools upon their issuing the appropriate message. Other examples for communication frameworks are ToolTalk [Juli94] (used in Merlin), SoftBench BMS [Cag90], Field [Reis90] and CORBA [Grou92]. These systems are on different levels of abstraction and they use different basic techniques, nevertheless they can all be used to build communication managers in open architectures.

One major task in such an architecture is to define communication interfaces for its components and protocols that define the proper use of the services provided. In order to achieve an open architecture, many applications use message-passing oriented com-

---

1. DEC Friendly Unified Software Environment
munication combined with centralised broadcast services. This approach facilitates easy substitution of tools.

Existing message passing protocols typically use the following basic model: messages can be divided into requests and notifications. A request is sent to a tool in order to use functionality provided by that tool. The requesting component is then waiting for a response containing either the result of the requested function or a reason for its failure. A notice is sent whenever a tool has performed some activity which might be of interest to other components (e.g. creating a new file, or opening another file for editing).

In the case of a process centred environment the simple protocol sketched above has to be extended by negotiation functionality: the activity leading up to a notification has to be checked for permission to proceed by Process Management before it is performed.

Communication protocols for requests and notifications are used to preserve consistency between different representations of data maintained by various components in a PSEE. Failures during communication (e.g. due to network errors) can result in inconsistencies between these data representations. The communication protocol has therefore to be defined in such a way that failures can be detected and recovered.

5.1.7 Tools

Various tools are used to perform different activities in different phases of the software development process. Examples of such software development tools (SDTs) are (graphical) editors, browser, pager, debugger, compiler and analyzer. Since one of the main tasks of a PSEE is the coordination of activities performed by multiple users, the environment has to control invocation and termination of SDTs.

SDTs can be classified into two categories depending on whether or not their application can have influence on the course of the executed software development process, i.e. the state of the current project. An SDT can influence the state of the current project if it is able to modify (or even create) documents. These tools we refer to as process relevant tools (e.g. editors) in contrast to process irrelevant tools (like viewer and browser).

When a process relevant SDT is executed, the changes to the state of the current development project have to be determined. For most non-interactive tools which typically are executed in batch mode (e.g. compiler, linker, analyzer) this may be done automatically through examination of exit codes which indicates the result of the performed activity. When an interactive SDT is applied the user may have to inform the PSEE about the effects of the performed activity on the state of the current project, e.g. a state transition for an implementation module from not_yet_implemented to implemented after the termination of an edit activity.

Software engineers who have become accustomed to a specific working environment resist changing to a completely new environment, especially before its advantage has
been conclusively demonstrated. Thus best user acceptance can be achieved for a PSEE if it has an open system architecture which supports integration of existing SDTs.

So far we have sketched integration of SDTs on the granularity level of documents which is rather coarse. A more fine grained granularity of integration is desirable for SDTs which work on documents including information that overlaps with parts of the project data. This is typically the case for SDTs applied in early phases of the software development process. These maintain documents which include information about other documents and their relationships. For example in a design document each described module corresponds to an implementation module in the user’s workspace. Interdocument relationships should be controlled even in those SDTs used in later phases of a software development process, such as in import/export sections of implementation modules.

Fine grained tool integration facilitates incremental and intertwined software development processes and avoids inconsistencies between representations of documents and workspaces with respect to project information.

Suitable SDTs for fine grained integration have to fulfil special requirements in that they have to be syntax directed, i.e. they must work on high-level data structures rather than on plain ASCII files. In addition, they have to provide sophisticated communication services in order to couple their local process models with the global process model of the PSEE.

Consequently, whenever a user invokes an interaction that modifies inter-document information the PSEE is requested whether this interaction conforms to the global process model. If this is the case the modification is performed in the subject document as well as in the project information. On the other hand, if the interaction is not allowed an error message is displayed to the user. It should be noted that a variety of suitable existing SDTs can be integrated in a PSEE by the application of envelopes. An envelope is an adapter between different communication interfaces.

There is no doubt that the construction of SDTs with sophisticated functionalities such as those described above is a complex task. As a consequence, our current research activity is focused on automatic tool generation based on a dedicated high-level tool specification language which facilitates abstract description of sophisticated communication services [Emme95].

5.2 Architectures for Distributed PSEEs

5.2.1 Determinant Requirements on Architectures for Distributed PSEEs

Many of the available PSEEs provide a client/server model for process enactment. One central process server is used to support many users who are connected to this server. This kind of client/server model, provided for Local Area Networks (LANs), will fail for distributed software processes for three reasons:
Large scale software processes are not executed in the centralised client/server organisation as described above. Usually, such projects consist of distributed teams in different companies. Different companies have different software processes which sometimes (e.g. if they involve sensitive information) must not be visible to all other companies, teams or persons. Therefore, knowledge about software development in such projects is also distributed over the involved companies. Only some small interfaces to exchange documents and process data are required. Referring to this kind of software development we can define two requirements which must be supported by a distributed PSEE:

**R1.** (Requirement for hierarchical or distributed process organisation). The process model itself must be organised in a distributed, hierarchical fashion, i.e. it realises a client/server model itself.

**R2.** (Requirement for distributed process data). In order to avoid misuse of process data and documents, both should be stored at the site (company/team) to which they belong. This data may be made visible to other sites to allow for necessary data exchange.

Central process engines serving a few software developers (as their clients) in a LAN is considered to be ‘state of the art’. Hundreds or more of software developers, connected to one central process engine, will suffer performance problems because a lot of information has to be exchanged between the clients and the server, i.e. the central server becomes the bottleneck.

**R3.** (Requirement for distributed process management). Therefore more than one process engine must be provided to ensure adequate performance. They may be working on a central, their own, or a partially distributed repository.

In geographically dispersed teams a further communication aspect arises: that of the information exchange via a Wide Area Network (WAN). If any request which belongs to some process information has to be exchanged via a WAN, the process engine will slow down because of delay times, net problems, etc.

**R4.** (Requirement for distributed process engines). Distribution of the process management over the sites involved in the software process is therefore required. These process engines support either users or further process engines (which may be distributed). The data exchange between the process engines is then reduced to the cases where data which belong to other machines on other sites are accessed or touched.

### 5.2.2 Architectural Alternatives for Distributed PSEEs

The requirements defined in Section 5.2.1 can be mapped on the axes in the Figure 5.4:

From this model the following four alternative architectures can be derived which more or less meet the requirements defined in Section 5.2.1:

- **Architecture I.** Central process engines, central process data.
- **Architecture II.** Distributed process engines, central process data.
Architecture III. Distributed process engines, partially distributed process data.

Architecture IV. Distributed process engines, distributed process data.

The selected architectures can be described as follows:

- Architecture III. Distributed process engines, partially distributed process data.
- Architecture IV. Distributed process engines, distributed process data.

The selected architectures can be described as follows:

2. In the applied graphical notation rectangles are used to represent modules or subsystems and directed edges denote used-relationships between modules or subsystems.
is distributed (assuming client/server support by the process model), but a central process database violates requirement R2. In spite of this, the requirements to support a software process in a LAN are fulfilled. What is new in this architecture is that the common repository is used by the different process engines to communicate with each other about process changes that may cause updates of user environments supported by those process engines. This communication is based on a notification mechanism provided by the common repository which notifies process engines when changes of the common process data occur.

**Figure 5.7 Architecture III**

Architecture III meets all requirements defined in Section 5.2.1, because additional local process databases have been added to the process engines. As in architecture II, there is a common process data base visible to all process engines which stores common process data and serves as the communication platform for the process engines.

The architecture in Figure 5.8 (Architecture IV) also meets all requirements defined in Section 5.2.1. The difference between it and Architecture III is that no global process database is used to exchange data. This is done by introducing a communication link between the process engines to enable the exchange of process data. However, this
architecture has to handle redundant data (stored in the local process data bases) which was held in the common process data base in Architecture III.

Figure 5.8 Architecture IV

5.3 Example Architecture: The Distributed PSEE Merlin

This section describes the architecture of the PSEE Merlin [Junk94] which has been developed in Germany at the Universities of Paderborn and Dortmund. The first subsection reflects the architectural alternatives given in the previous section and sketches how instances of basic components of the Merlin environment interact. Section 5.3.2 describes the Merlin architecture in more detail and gives further information on the realisation of each component.

5.3.1 Instance View on the Merlin Architecture

The Merlin architecture for a distributed PSEE is a combination of Architectures III and IV (see Figure 5.9). Hierarchical client/server architectures can be built using the architectures described above by requiring client/server behaviour of the process and distributed process engines/data. The leaves of the tree which is described by this hierarchy are clients which serve one or more software developers, i.e. their corresponding working contexts. This is exactly Architecture I.

Architectures II, III and IV are able to fulfil the requirement of supporting a PSEE in a LAN and a WAN. Note that Architecture II is a special case of Architecture III (without the local project data bases), but this architecture does not meet the requirement R2. The functionality provided by Architectures III and IV is the same; only the performance may differ. For the Merlin PSEE we chose Architecture III and added communication links between the different process engines (see alternative IV), in order to increase efficiency of change propagations.

5.3.2 Type View on the Merlin Architecture

Figure 5.10 represents the type view of the Merlin architecture. In this representation directed edges denote used-relationships between the different components. The five
main subsystems \textit{Workspace, Process Management, Tools, Communication} and \textit{Repository} correspond to basic components introduced in Section 5.1.1.

\textbf{Figure 5.9} Merlin Architecture (Instance View)

\textbf{Figure 5.10} Merlin Architecture (Type View)

The following sections describe further details of implementation for each subsystem in the Merlin architecture.
5.3 Example Architecture: The Distributed PSEE Merlin

5.3.2.1 Subsystem Dialog Management

For each developer, component Dialog Control controls the invocation and termination of all applied tools and the display of his/her specific workspace, a so-called working context. It is connected with the corresponding process engine via a communication link using the broadcast server. The communication link is used to receive updates of the developers’ working context, to exchange menu information and to inform the process management about every activity the developer is going to perform.

Information about the actual working context is maintained as hypertext in component Workspace Representation which encapsulates an attributed graph. In this data structure documents are represented as nodes while relationships between documents are represented as directed edges between nodes. Further information about possible activities on specific documents or access rights are stored as node attributes.

As an example of the visualisation of a working context in Merlin consider Figure 5.11. The sample working context represents the view of user Miller in his/her current role programmer on the information concerning project merlin_demo_project (see the top line of the Working Context Window). It displays the documents to be accessed and their interdependencies and the activities that may be or have to be performed on each document.

Documents are represented as rectangles that have context-sensitive menus attached which contain all performable activities on the corresponding documents. Inter-document relationships are described by labelled arrows between rectangles. Selection of a menu item triggers the execution of an activity and hence the invocation of the corresponding tool. Possible menu items are displayed in Figure 5.11 where programmer Miller has read access to the specification documents m1_spec and m2_spec (by ascii_pager and ascii_printer) as well as read/write access to the implementation document m1_c.

After the execution of a modifying activity the new state of the corresponding document has to be set by the user. This may be done by the application of a selection window which offers a list of all possible new states for the document. As an example we assume that Miller has selected the activity editor for m1_c. In this case two additional windows appear on the screen, one for editing m1_c and the second to select the new state.

In addition, the subsystem Dialog Management provides generic user interface services including various dialog objects which enable interaction between the developer and Process Management, e.g. to support the transaction manager by selecting a specific transaction type for an activity and interactive resolution of conflicts (see Section 6.4.2).

5.3.2.2 Subsystem Tools

This subsystem includes all kinds of tools integrated in the Merlin PSEE. Highly integrated SDTs can be generated from specifications [Emme95] or adapted for their application in Merlin by using ToolTalk [Juli94] envelopes. Tools are classified into tool
classes corresponding to their application purposes (e.g. ASCII-editors, debugger, pager). Each developer can customise his or her workspace to use the tools he/she prefers. The representation of documents maintained by the different tools varies from plain text files stored in the Unix filesystem to abstract syntax graphs stored in non-standard database systems like GRAS [Lewe88] or object management systems such as GemStone [Bret89] or O₂ [Deux91].

Merlin provides a number of tools with special functionality such as Mail and Query. The mail tool is used for communication purposes between the different users of the PSEE. It supports special features like abstract addressees (e.g. tester of module m1.c) and message redirection.
The user can retrieve additional information about the process, project history or impact analysis using the tool *Query*. This tool translates user queries into queries that can be understood by the reasoning component (subsystem *Process Management*) and transforms answers into a comprehensible format.

### 5.3.2.3 Subsystem Process Management

The process engine (module *PE-Interpreter*) is implemented as an interpreter for a rule-based language similar to PROLOG. For application in a PSEE the language PROLOG has been extended by special predicates, e.g. for communication purposes or to maintain persistent process data (rules and facts).

As depicted in Figure 5.9 there is a separate instance of a process engine for each Merlin user. Synchronisation between different process engines is performed via a common repository and the notification mechanisms provided by the Merlin communication subsystem.

The Software Process Program is a collection of rules, facts and predicates and is separated in three layers as explained in Section 5.1.3. The two subcomponents *Transaction Manager* and *Version Manager* are also implemented in PROLOG rules and actually belong to the Cooperation Model of the Software Process Program. Nevertheless, as a key component of the Process Management we decided to bring out their logical position in the Merlin architecture clearly in Figure 5.10.

The *Reasoning* component uses the *PE-Interpreter* to evaluate impact analysis and retrieve information about project history. Information about project history is also stored as facts in the *Project-Layer* of the SPP.

Specifications of processes in the high level PML ESCAPE [Junk95] are translated into executable PROLOG programs by means of the *Process Modelling* component. Furthermore information about concrete projects like responsibilities and roles can be modified.

### 5.3.2.4 Subsystem Repository

Merlin uses an heterogeneous, distributed repository to maintain persistent *Software Process Programs*. The object management system O₂ [Deux91] is used to store and retrieve all information residing in the *Project-Layer*. In O₂ rules and facts are distributed in so called *rule clusters* or *fact clusters*. Such clusters can be dynamically made visible or invisible for certain process engines.

The mechanism for ACID transactions provided by O₂ is used as platform for the sophisticated Transaction Manager in Merlin (see Section 6.4.2). Furthermore O₂ provides an advanced flexible mechanism for versioning of complex objects which is exploited in the Merlin version and configuration management. The object management system O₂ has been used also for implementing the SPADE-1 prototype.

Currently, the Unix filesystem is used to store all information in the *Process-Layer* and the *Cooperation Model* of the *Software Process Program*. Tools that have been
integrated in Merlin use for example the non-standard database system GRAS[Lewe88]
or GemStone[Bret89] for persistent representation of documents.

5.3.2.5 Subsystem Communication

Basically, the Communication subsystem includes two components. Messages between
distributed) instances of Process Management and Dialog Management components
are exchanged using the Broadcast Server. The Broadcast Server maintains a list of all
communication clients which are currently active and redistributes incoming messages
to all clients of interest. In the existing Merlin prototype the Broadcast Server is imple-
mented using TCP/IP communication services. There will shortly be an extended
implementation based on open ToolTalk [Juli94] protocols.

The Communication Interface component is used by all those other components in the
Merlin architecture which are designed to exchange messages via the Broadcast Server.
It provides basic functionality to send and receive synchronous and asynchronous mes-
sages.