Chapter 6
Cooperation Control in PSEE

Editor: Claude Godart

Contributors: Noureddine Belkhatir, Antonio Carzaniga, Jacky Estublier, Elisabetta Di Nitto, Jens Jahnke, Patricia Lago, Wilhelm Schaefer, Hala Skaf

6.1 Introduction

6.1.1 Objective

Cooperation is "acting together in a coordinated way in the pursuit of shared goals". Cooperation has numerous benefits: *efficiency*, where cooperation minimises the effort required to achieve the goal; *enhanced capability*, where a group can achieve a goal not possible for one person to achieve; *synergy*, where the cooperating partners can together achieve a different order of result from that achievable separately. Cooperation has also one main disadvantage: there is always an associated energy and time *overhead.*

Current software processes are cooperative and our objective in this chapter is not to propose new cooperation patterns. It is, rather, to describe and support current cooperative software processes, with an emphasis on the problems of consistency and integrity maintenance of software artifacts.

In fact, software artifact consistency and integrity maintenance on the one hand, and cooperation support on the other hand, are somewhere antagonistic. Classically, consistency and correctness are achieved by isolating the processes which execute in parallel (typically, most traditional database transaction protocols imposes isolated execution of processes). However, cooperation means that it is beneficial for the people involved in the processes which execute in parallel to interact. Thus, a challenge when building a PSEE to support software processes is to enable cooperation while continuing to achieve correctness.

Three approaches can be considered. A cooperation policy can be based on:

a) the responsibility of human agents alone, as it is the case in most current software development applications,
b) the knowledge included in the software process model without identifying any specific knowledge related to cooperation support. This makes the hypothesis that all the interaction cases are forecast. ADELE and SPADE appear in this category.
c) some predefined strategies, not depending on a particular process and which can be reused from one process to another. This is the transaction paradigm as in COO, and Merlin for example.

J.C. Derniame, B.A. Kaba, and D. Wastell (Eds.): Software Process, LNCS 1500, pp. 117-164, 1999.

In general, a cooperation policy is a combination of these approaches: transactions make the hypothesis that processes can be launched at the appropriate time (based on some knowledge of the process) and knowledge can be organised in reusable libraries (which can be considered as predefined strategies as transaction models are).

In the rest of this chapter, we illustrate these approaches by means of a common example: COO and Merlin are representative of the transaction paradigm, ADELE and SPADE of the purely process based approach. This objective is achieved, on the one hand by a short description of the architecture of each system considered, and by the implementation of the example introduced in the following section (Section 6.1.2) in each system.

6.1.2 An Illustrative Example

Let us consider a supposed simple software process (see Figure 6.1). The objective of the task is to produce the code of a module B which depends on a module A. Module A and module B are developed by two groups of people who want to work in synergy. This is a very common situation in software processes and that defines a pattern which can apply to different artifacts at different levels of a software life cycle.

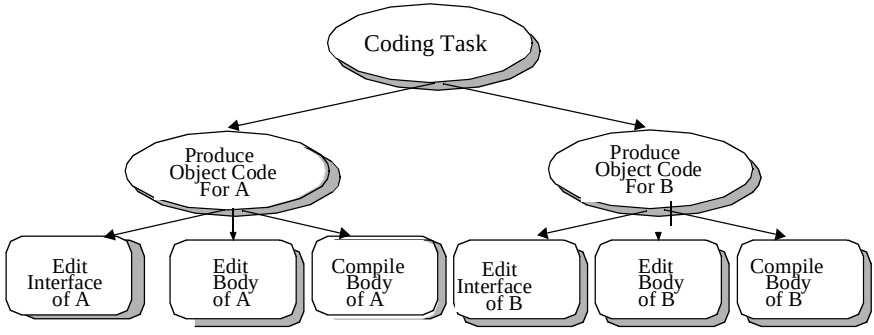

Figure 6.1 A simple process

This process is hierarchically organised. The root process represents the whole process which breaks down into sub-processes. The leaves are atomic processes (which execute atomically in isolation), and the upper levels are compound processes.

The objective of this *coding_task* process is to “produce the object code for a module A and the object code for a module B”. Thus, *coding_task* splits into two sub-processes: *produce_object_code (A)* and *produce_object_code (B)*. Each sub-process executes as a consistent combination of one or several occurrences of *edit_interface*, *edit_body* and *compile* applied to A or B. Each leaf process is an abstraction for *read* and *write* oper-

ations. Lets now suppose that the body of the module B (B_b) depends on the interface of the module A (A_i). This can be interpreted as: *edit_body(B)* is an abstraction for *read* (A_i), and *write*(B_b). In this context, it is clear that the two operations *edit_interface(A)* and *edit_body(B)* are concurrent when accessing A_i and that the produce_object_code (B) sub-process must at least enforce a rule which ensures the body of B (B_b) is edited with the final (last) value of the interface of A (A_i). This rule describes a part of our knowledge about the *coding_task* process and a correct execution of *coding_task* is always an execution of the six (sub-)processes: *edit_interface(A)*, *edit_body(A)*, *compile_body(A)*, *edit_interface(B)*, *edit_body(B)*, c*ompile_body(B)*, which respects the policy (rules) just defined.

Let us now consider three scenarios on the following pages.

Scenario 1 is a correct execution of *coding_task* in which *produce_object_code(A)* and *produce_object_code(B)* execute without interaction. The only object in shared by the two processes, the interface of A, is accessed first by *produce_object_code(A)*, then by *produce_object_code(B)*. Clearly, assuming the atomicity of leaf sub-processes, this execution is equivalent to the serial execution of *produce_object_code(A)* followed by *produce_object_code(B).*

Table 6.1 Scenario 1: correct and without interaction execution of coding_task

produce object code A	**produce object code B**
edit_interface(A)	
edit_body(A)	
compile_body(A)	
edit_interface(A)	
	edit_interface(B)
edit_body(A)	
compile_body(A)	
edit_body(A)	
compile_body(A)	
	edit_body(B)
	compile_body(B)
	edit_body(B)
	compile_body(B)

In contrast, Scenario 2 depicts an execution in which *produce_object_code(B)* reads the interface of A at the same time it is being modified by *produce_object_code(A*). In fact, *produce_object_code(B)* sees an intermediate value of *produce_object_code(A)*: the two processes interact. Clearly, this schedule can be non serialisable. Nevertheless, we can verify that this scenario is also correct. This is based on the following reasoning. Firstly, *produce_object_code(B)* reads the final value of the interface of A, since the last *edit_body(B)* appears after the last *edit_interface(A)*. Secondly the last *edit_body(B); compile_body(B)* sequence compensates all the previous ones (in the sense that its result is the same as if the previous sequences did not occur). Thus, even if the value of the body of B is inconsistent at *point 1* because it is based on an intermediate value of the interface of A, the final value of the body of B, at *point 2* is consistent with module A because it is based on the final value of the interface of A.

Table 6.2 Scenario 2: correct and with interactions execution of coding_task

produce object code A	**produce object code B**
edit_interface(A)	
	edit_interface(B)
	edit_body(B) <- point 1
	compile_body(B)
edit_body(A)	
compile_body(A)	
edit_interface(A)	
edit_body(A)	
compile_body(A)	
edit_interface(A)	
	edit_body(B)
edit_body(A)	
compile_body(A)	
	compile_body(B) <-point 2

Scenario 3 demonstrates that some sequences of the same atomic processes can be incorrect: the last value of the interface of A read by *edit_body(B)* is not the final one. The interface of A has been changed since the last edition of the body of B.

Table 6.3 Scenario 3: incorrect execution of coding_task

produce object code A	**produce object code B**
edit_interface(A)	
	edit_interface(B)
	edit_body(B)
	compile_body(B)
edit_body(A)	
compile_body(A)	
edit_interface(A)	
edit_body(A)	
compile_body(A)	
	edit_body(B)
edit_interface	
edit_body(A)	
compile_body(A)	
	compile_body(B)

We have pointed out two executions of the same process which are both sequences of the same atomic activities and which are both correct. However, these schedules are not equivalent. The second allows interactions to occur between *produce_object_code (A)* and *produce_object_code(B)* while the first does not. We think that interactions between processes must generally be supported. In fact, interactions between activities also imply interactions between people which are generally positive in social processes such as software development processes. Nevertheless, it is also clear that in some cases interactions can be considered as undesirable, and visibility of intermediate results must

be prohibited. This is especially the case when intermediate results are used by processes which cannot be easily compensated (as an example, putting an insufficiently tested product on the market can have huge consequences). In such case, an isolated execution as in Scenario 1 is the most commonly adopted solution.

This example illustrates a simple case of positive interaction, of *cooperation*. Others are characterised in the rest of this chapter.

6.1.3 Organisation of the Chapter

The following section (Section 6.2) makes a survey of traditional and advanced transaction models. It identifies their limits and advantages with regards to our process characteristics: uncertain duration (from hours to months), uncertain developments (i.e. activities that are unpredictable at the beginning), and interaction with other concurrent activities. The motive is to introduce the problems related to consistency maintenance in general and the basis on which "predefined strategies approaches" are founded. Section 6.3 addresses the issue of how cooperation control impacts the PSEE architecture. Section 6.4 describes the current work on the topic in the Promoter Working Group. The last section, (Section 6.5) concludes.

6.2 Moving from Traditional to Advanced Applications

The definition of *transaction* for advanced applications changes, mainly due to human-interaction. It is transformed from a pre-programmed implementation to a variable working session. Accordingly, a fundamental difference from traditional transactions is represented by characteristics like non-atomicity and non-isolated execution.

Before deepening this evolution, we are reminded of the basic properties, called ACID, of traditional transactions. [Bern87a] provides a sound introduction to concurrency control and transaction processing.

6.2.1 ACID Properties

An important concept for traditional *database transactions* is the ACID properties (Atomicity, Consistency, Isolation and Durability).

Atomicity: A transaction is an atomic unit of processing. It either executes in its entirety or not at all. This is the "all or nothing" law.

Consistency: A transaction execution results in a database state that satisfies all the consistency constraints of the application; that is, if the program has functioned according to its specification[1].

Isolation: A transaction should not make its modification visible to other transactions until it is committed, i.e, until its effects have been permanently recorded in the database.

1. This assumes the specification is correct and complete.

Durability: Once a transaction results in a new database, the performed modification must never be lost because of system failure.

These properties are too strict for software processes. This is especially due to the *atomicity* and *isolation* properties.

6.2.2 From ACID to Non-ACID

Atomicity and Isolation have two implications. First, it limits cooperation: if requested data is kept private or locked by one long transaction until it terminates, other concurrently running transactions will be forced to wait for its termination. Thus, it prevents data from being freely exchanged among humans, and made accessible as soon as possible.

Second, if a transaction fails[2], the work done inside its context should be undone. A lot of work could be thrown away, although not influenced by the causes of failure. Further, if cooperation took place with the failed transaction, transactions cooperating with it could possibly be aborted in cascade. Thus, a requirement for long transactions allowing cooperation is to separate work done privately from work done cooperatively (and thus influenced) by many transactions. In fact, in the case of failure, it should be possible to un-do one transaction's (private) work at a fine-grained level, and to restart it, without causing related, but not affected, cooperators to clash.

6.2.3 From Flat to Nested

Flat transactions are allowed to execute atomic data accesses. *Nested transactions* may also start new (sub-) transactions, that is decompose themselves into transaction hierarchies. Nested transaction are well adapted to hierarchically organised processes such as software processes.

6.2.4 From Closed to Open

Nested transactions in general refers to closed-nested transactions, as opposed to open nested transactions. *Closed-nested transactions* have been mainly introduced to limit the impact of a logical or physical failure to a sub-transaction. Atomicity and isolation of (sub-)transaction executions is preserved. *Opened-nested transactions* relax isolation and can relax atomicity by allowing partial results to be observed outside the transaction before its completion. To maintain consistency of execution, although the isolation principle is not preserved, the semantics of high-level operations are exploited. On account if the long duration, uncertainty and interactive nature of our processes, openness is preferred.

2. Failure may be either user-driven (e.g., certain activity is cancelled from a project), or process-driven (e.g., due to conflicts with other activities).

6.2.5 Hierarchical versus Layered

When a transaction splits into sub-transactions, a question arises: *do we need to preserve consistency globally or locally?* Can we have one scheduler per (sub-)transaction or must we have a global scheduler for all the nested transactions? Theory demonstrates that we can have one global scheduler or, with some restrictions on the organisation of the transaction, one scheduler per level. In the first case, we refer to (vertical) *hierarchical transaction* breakdown, in the second case, we refer to (horizontal) layered transaction management. We think that *layered transactions* are too rigid because of the dynamism of our processes.

6.2.6 Homogeneous versus Heterogeneous

Homogeneity refers to the way transactions are scheduled, or more generally, how transaction behaviour is managed. A global scheduler can either integrate local schedulers which implement the same protocol, or schedulers which implement different protocols. In the first case, the scheduler is homogeneous, in the second case, it is heterogeneous. It must be also noted that not all combinations are possible. Clearly, in the case of a layered transaction, different protocols can be implemented at different levels without restriction. In the case of one heterogeneous global protocol, interferences between sub-protocols must be managed.

6.2.7 From Transient to Persistent

Persistence is the ability of objects to persist through different program invocations [Atki96]. Accordingly, objects may be either transient or persistent. Transient objects are only valid inside the program scope, and they are lost once the program terminates. Persistent objects are stored outside the program and survive updates. In other words, they persist across program execution, system crashes, and even media crashes. These objects are the recoverable data of the management system, and are shared, accessed and updated across programs.

Traditionally, transaction structures are in main memory, i.e., they are not persistent. Thus, if a failure occurs, the transaction rolls back. This is not acceptable for long term transactions: transaction structures must persist in such a way that, in case of a failure, the process engine is able to restart from a state close to the failure state.

6.2.8 Available Advanced Transaction Models

This section reviews several representative advanced transaction models: closed nested transactions, split transactions, cooperative transactions, layered transactions, and the S-transaction model. This is done through the definition of a transaction model in terms of its three components: structure, primitives and protocol.

6.2.8.1 Advanced Transaction Models: Classification

To reason about the transaction requirements for a particular application class, advanced transaction models are first analysed at the design level. A *transaction model* is a representation of the elements relevant for transaction definition and execution. It consists of three main parts:

Transaction structure: it defines how single transaction types are defined, and how sets of transactions are syntactically related together. Transactions may be ***ACID*** (i.e., atomic units that fulfil ACIDity); ***Compensating*** (i.e., transactions that can semantically undo the effects of an associated transaction after this has been committed [Wach92]); ***Contingent*** (i.e., alternative transactions that are executed if a given transaction fails); or ***Vital*** (i.e., transactions whose termination influences the termination of the related transaction. For example, if a sub-transaction has a *vital relation* with the parent, then its abort causes the parent abort).

Transaction primitives: which are the basic operations, i.e., to manage internal behaviour (e.g., creation, execution, termination), and external behaviour with respect to other cooperating transactions (e.g., communication, data transfer, data access synchronisation).

Transaction protocol: which is the behaviour of a set of transactions viewed as a global unit that execute concurrently and need to cooperate. Related issues are concurrency control and recovery.

In summary, the transaction structure reflects the organisation/work breakdown, while a transaction protocol defines how transactions can interact, and how they can access data.

6.2.8.2 Closed Nested Transactions

The enhancement of flat to nested transactions introduces two main important advantages [Moss85]:

1) execution parallelism internal to a non-leaf transaction (intra-parallelism), and
2) finer control over failures and errors, thus achieving recovery guided by semantic information.

Transaction Structure

Transactions can have infinite levels of nesting. The semantics is that of decomposing a transaction into a number of sub-transactions, the overall structure being a tree (or hierarchy).

Transaction Primitives

Apart from traditional atomic operations for transaction creation and termination (Commit and Abort), and for data access (Read, Write, Update and Creation), transactions are allowed to invoke atomic transactions as well. By invoking a transaction from within its scope, a sub-transaction is created and the nesting is implicitly achieved.

Transaction Protocol

Several protocols have been modelled to achieve transaction synchronisation. The way transactions declare their work intentions (i.e., how they reserve data) differs from model to model. Their suitability depends on the kind of application the model is used for. The main techniques are based on locking and time-stamps.

Furthermore, locking can be exclusive or refined into read and write mode. In both cases, to preserve isolation, the parent inherits the lock mode (at child commit) and stores it in its own place-holder. The effect is to grant data access only to descendants: if a transaction tries to lock some data, it is allowed to do so only if either the data has no lock at all (i.e., is free) or its parent already has the lock in its place-holder.

If a child transaction aborts, locks are simply released and not inherited by the parent as no effects on data persist.

6.2.8.3 Split Transactions

Split transactions propose a solution for restructuring in-progress transactions, still preserving consistent concurrent data access. Split transactions support open-ended activities like CAD/CAM projects and software development. A practical example of this model's usage is in user-controlled processes, where users determine what database operations to perform dynamically, during execution. In real applications, split transactions are a natural means of committing some work early or dividing on-going work among several co-workers. Further, a split transaction can hand over results for a co-worker to integrate into his/her own ongoing task.

Advantages of this model are:

a) adapting recovery effort: because split transactions can abort without affecting the result of the original (splitting) transaction.
b) reducing isolation: by committing part of a transaction, resources can be released whenever needed, thus achieving higher concurrency.

Transaction structure

The original (simple) model assumes transactions as composed of a sequence of operations, similar to ACID transactions, i.e., abstracted into Read and Write operations, starting with Begin, and ending with either Abort or Commit.

To include parallelism, the model has been extended with nesting, where a set of top-level transactions may be executed concurrently. Later on, super-transactions have been included (see [Hard93]): a super-transaction includes two or more independent transactions, and makes them appear atomic to transactions outside the super-transaction itself.

Transaction primitives

Primitives peculiar to split transactions are Split and Join [Kais92]. The Split operation divides the on-going transaction into two (or more) serialisable new transactions, and assigns its resources among the resulting transactions. Thus, Split and Begin represent the set of initiation operations by means. The Join operation merges two or more transactions that are serialisable with respect to each other, into a single transaction as

if they had always been part of the same transaction, and all their work is now committed or aborted together. Thus, Join, Abort and Commit represent the set of termination operations.

Transaction protocol

The major purpose of splitting is to commit one of the resulting transactions to reveal useful results from the original transaction. An example is shown in Figure 6.2, where transaction T initiates a Split operation that results in two new transactions (A and B) replacing T.

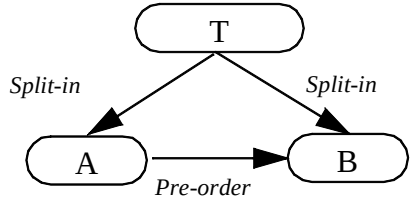

Figure 6.2 Split transaction T into transactions A and B

Of course, when splitting, consistency on data must be preserved. For instance, supposing in the example that transaction A and B fulfil properties:

$$WriteSet(A) \cap WriteSet(B) \subseteq WriteLast(B)$$

$$ReadSet(A) \cap WriteSet(B) = \varnothing$$

$$WriteSet(A) \cap ReadSet(B) = SharedSet$$

B must be executed after A, as the work context of B depends on results in the work context of A. In other words, for property 1, any data written by both A and B is written last by B (i.e. A cannot overwrite B's output), for property 2, A does not have visibility on data which is changed by B (and thus does not depend on B), while for property 3, B has visibility of data changed by A.

To allow A and B to be independent, the following properties should hold:

$$WriteSet(A) \cap WriteSet(B) = \varnothing$$

$$ReadSet(A) \cap WriteSet(B) = \varnothing$$

$$WriteSet(A) \cap ReadSet(B) = \varnothing$$

6.2.8.4 Cooperative Transactions

The concept of cooperative transaction was introduced in [Nodi92] to support a higher level cooperation than in traditional transaction models. This model substitutes the notion of correctness defined by serialisability, with a notion of user-defined correctness by means of Finite State Automata. In this way, cooperative transactions can use different correctness criteria (the most suited for their own purposes). Further, isolation is not required, and hierarchy allows both close cooperation to take place, and easier management for long-lived transactions.

Transaction structure

Transactions are organised in a rooted hierarchy, where internal nodes represent groups of transactions that cooperate to perform a single task, while external (leaf) nodes are associated with individual designers. A transaction group (TG) is thus a collection of cooperative transactions that cooperate to perform a specific task, or other TGs. It can be seen as the abstraction of a task, undertaken by its members. With this viewpoint, TGs are spheres of control over children. A cooperative transaction (CT) is a program that issues a stream of operations to the underlying database. It can be defined as the sphere of control over actions on data.

Transaction primitives

In terms of primitives this model does not define new constructs (using the traditional begin and commit/abort), but define a particular way in which they are managed. In fact, primitives define atomic actions on a single object by a single CT member of a TG. Atomicity in this case means that primitives cannot invoke other primitives, and are defined either on objects (e.g., Read or Write), or on their interface.

During its lifetime, a CT issues primitives on objects in its TG. There primitives are executed by the TG once it determines that they conform to its correctness specification. If not, primitives are rejected, i.e., individually aborted. This means that primitives are submitted to the TG which uses the correctness criteria to abort those which are not valid and accept those which are valid.

Transaction protocol

Transaction behaviour is modelled by a correctness specification (or criteria) of TGs on each single CT.

A correctness specification is defined in terms of:

a) conflicts, identifying primitives that are not allowed to execute concurrently,
b) patterns, defining sequences of primitives that must occur,
c) triggers, taking actions when a request by a CT begins or terminates.

A TG specification describes its valid history; the allowed sequence of primitive invocations issued by its member CTs. Thus a TG identifies the sphere of cooperation that takes place among its component CTs. Thus, correctness is defined in a top-down fashion, pertaining the TG members. Moreover, a TG history is correct if it satisfies its

own criteria, and the histories of its member CTs are correct, i.e., if it conforms to all the patterns and contains no conflicts.

A certain transaction protocol is defined for each TG, by the relative specification. Two alternative ways can be used to enforce a protocol: optimistic, where CTs issue their entire sequence of primitives and thereafter this is checked against the protocol, or on-line, where each primitive is controlled when issued. The latter is a better solution, in that it avoids potential wasted effort. Nevertheless, to be feasible, any solution should be able to recognise correct histories as well as valid prefixes of correct histories. The difficulty arises because specifications can vary over time as members are added or removed from a TG, and more generally because correctness rests on definitions of correct histories and is not easy to prove.

6.2.8.5 Layered Transaction Model

Layered transactions were introduced to handle applications that are naturally organised in layers like federated databases (four database layers that collect data into a federation of local databases, each organised in clusters of objects), operating system transactions (ISO/OSI seven layers), and federated DBMSs. For instance, the latter is a collection of DBMSs that cooperate loosely but are not fully integrated. It therefore involves the coexistence of global transactions, divided into local sub-transactions on the different DBMSs, and local (independently issued) transactions active on the local sites. In such an architecture, global transactions and local sub-transactions may have a non-serialisable schedule, while the coexistence with local independent transactions causes pseudo-conflicts, i.e., conflicts at the database level.

The layered transaction model is a variant of nested transactions where the nodes of the transaction tree define execution of operations at particular levels of abstraction in a layered system. L_i operations are treated as sub-transactions, and L_{i-1} operations as the actions that constitute sub-transactions. The key idea of layered concurrency control is to exploit the semantics of operations in level-specific conflict relations that reflect the commutativity of operations. In this way, a schedule at level L_i may be serialisable and operations may be executed in parallel whereas their implementation at level L_{i-1} is not, therefore needing a different concurrency control mechanism. Intra-transaction parallelism is achieved by handling sub-transactions at a lower level uniformly, regardless of whether they belong to different ancestors or to the same. In this respect, layered transactions are a specialisation of open nested transactions, where the tree of sub-transactions is balanced. In fact, (1) sibling transactions are executed independently from the parents, and (2) each top-level transaction is described at a fixed number of abstraction levels, equal to the number of layers of which the system is composed.

6.2.8.6 S-transaction Model

S-transactions (where “S” stands for semantic) have been developed for large-scale inter-organisational autonomous environments like international banking, to model communication protocols.

The main basic concept is autonomy: the ability to decide or choose a behaviour different from what is expected by the rest of the environment. There are various types of autonomy. For instance, design (or data definition) autonomy is when an organisation has full self-determination with respect to control the structure-type of interactions, i.e., when choosing hardware or software, when developing the database schema or other applications. S-transactions' applicability is immediately identified in the fields of banking, software engineering, and CIM.

Transaction structure

S-transactions are normal transactions associated with a corresponding compensating transaction. They preserve isolation and are defined in terms of local data (requested either to a remote S-transaction, or to the common database), and entry points (like attach channels) for continuations (e.g., sub-transaction activation or remote transactions' messages) and compensations, thus representing compensating transaction like an integral part.

Transaction primitives

Both semantic and compensating S-transactions are associated with traditional primitives Begin, Commit, and Abort. The difference lies in the semantics assigned to them. For instance, rules for compensation are defined to enact a compensating S-transactions that reverses the effects of the corresponding S-transaction, when the latter has failed.

Transaction protocol

The S-transaction model's potential is fully exploited by applications that manage large amounts of complex data, where interacting processes have semantics known in advance. Interaction takes place partly on the communication level (message exchange), and partly on the inter-operation level (partially ordered sequences of actions).

6.2.9 Summary and Analysis

The main characteristics of the models presented in Section 6.2.8, are sketched out in Table 6.4. It outlines the properties that each model directly addresses and aims to support. The objective is to guide the reader in choosing the right model for his/her purposes. Typical applications are also given For each model, possible answers to property coverage are:

yes, means fully supported,

yes:<comment>, means that the property is supported, limited to the context indicated in the comment,

no:<comment>, means that the property is not supported, except for the context indicated in the comment,

no, indicates that the property is not supported, neither in extended models based on the referred one.

No indication (i.e., an empty box) indicates that the “pure” model does not address the property, even though extended models based on it might do. For each model, the answer written in bold indicates the most significant property issued by the model..

Table 6.4 Properties covered by advanced models

Models	**con-currency control**	**co-operation**	**open-ending**	**user-defined correctness**	***applications***
nested-trans.	**yes**	yes: intra-trans.	no		CAD/CAM, SEE
split-trans.	yes	yes	**yes**		CAD/CAM, SEE
coopera-tive trans.	yes	**yes**	yes	yes	*design, SEE, CSCW, CE*
layered trans.	yes	no	no	**no: level-based trans-actions. commu-tativity**	operating systems, FDBMSs, OODBMSs
S-trans.	yes	no	yes	**yes**	SEE, int. banking, CIM

6.2.9.1 Available Issues and SEE Transactions

Transaction support in SEE’s has to fulfil basic requirements that reflect the behaviour of software activities. In software development, *cooperation* and *user independence* are key requirements. The first concerns the possibility of having control in a cooperative work environment. The second concerns the freedom to take decisions *on-the-fly* during development, in spite of control (and being therefore responsible for consequences). SEE's need therefore to embrace long-lived and cooperative transactions, to adequately control software processes.

Table 6.4 emphasises that concurrency control is managed by all models (at least as a contingent topic). Cooperation support, which is one of the most important topics in SEE transactions, has been undertaken by all of the described models, except S- and layered- transactions, which are specific to user-defined correctness. The last two properties are specific to advanced applications and have therefore only been addressed by recent models. Nested transactions do not support either of them. Layered transactions focus on user-defined correctness. The others support open-ending, which is the main topic of split transactions.

The most advanced support available in commercial systems is through nesting. Further, each system approaches cooperation from various viewpoints. For example, by allowing transactions to dynamically start, or by supporting undefined levels of nesting,

with an undefined number of sibling sub-transactions, or by transaction sharing among multiple-users.

In general, the nested transaction model suits SEE needs, as:

a) transaction hierarchy reflects a natural breakdown of work,

b) transaction nesting implements internal concurrency by allowing sub-transactions of the same parent transaction to run concurrently,

c) recovery is kept under control, by limiting undo to sub-transactions,

d) dynamic restructuring supports eventual open-ended activities, by allowing transaction adding and removal whenever a new transaction clustering is needed.

With regard to user independence, characteristics such as user-defined correctness and open-ending are not yet implemented in commercial systems. The main reason seems to be the need for a deeper knowledge of the exact definition of correctness in SEEs. In fact, being software activities controlled by humans, and due to the differences between processes, it is very difficult to define a standard and satisfiable set of common statements for consistency. A trend is to allow user-defined protocols to be written through rules (e.g., ECA rules), each associated with a transaction class. The extension of nesting with the cooperative transaction models seems to be suitable, although no commercial or academic results have reached a status stable enough for commercial implementation.

6.3 Impact of Cooperation Control on the Architecture of PSEE

This section provides more details to the architecture introduced in the previous chapter from the point of view of cooperation support. It makes no important presumptions nor restrictions on the architectural alternatives introduced in (Section 5.2.5, Chapter 5). Typically, the ADELE system (Section 6.4.3) has an architecture of type II, Merlin Section 6.4.2 of type III, COO (Section 6.4.1) also of type III but with some restrictions on the relationships between Process Engines. As depicted in Figure 6.3, we have identified five layers of abstraction which can be seen as service layers.

We will explain the impact of cooperation control on each of these layers.

The bottom layer is the *repository* layer. It mainly provides for object and schema management services and tool execution services. As emphasised earlier, in a modern PSEE numerous resources and software artifacts are managed. The modelling and management of these artifacts and their versions needs a powerful object manager. The properties of the underlying object model deeply influences the transaction models. Clearly, implementing the same transaction model on top of a different data model can be of different complexity: explicit links, composite objects, etc. are not easy to manage (to copy, to lock, etc.). On the other hand, distribution of the repository (clearly as in architecture alternative IV) generally increases the complexity of synchronisation, especially in the case of heterogeneous object management systems and even more so in the case of heterogeneous object model.

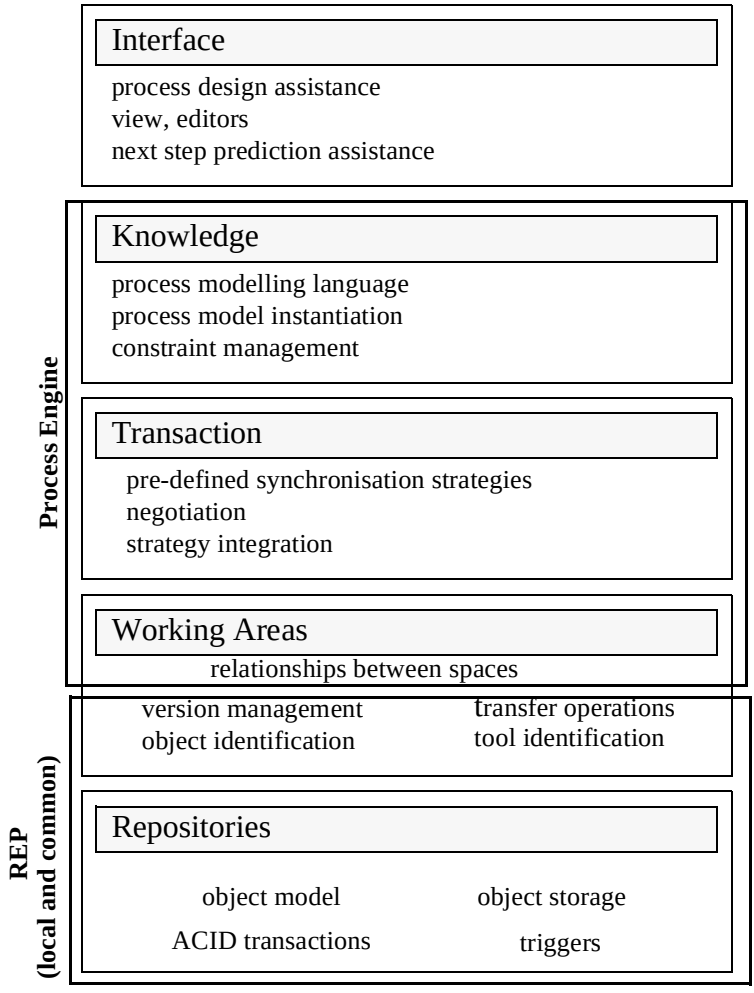

Figure 6.3 Transaction abstraction layers

In any multi-user environment, processes operate on a limited part of the object base. Typically, traditional transactions operate on object copies until commit; that is, copies define their working area. We therefore introduce the *working area* layer to reason about process isolation and object transfers without consistency considerations due to concurrent object accesses. Thus, the objective of this layer is limited to basic object transfers, i.e object identification. However, the characteristics of working areas can limit the supported transaction models. Typically, long duration and uncertainty of software processes imply that a working area must be able to outlive the sessions which create or modify it. In addition, the working area layer must be general enough to allow implementation of different transaction protocols simply by specializing and/or restricting transfers between working areas.

The *Predefined Strategies (*or *Transaction)* layer introduces consistency support and provides mechanisms to assert correctness of parallel processes. These mechanisms implements synchronisation strategies which are general enough to apply to different processes and to be predefined in a PSEE. This indicates that these mechanisms are mainly independent of the semantics of the processes which are controlled. Clearly, most traditional transaction models come into this category. For example, the 2PL protocol [Gray78] only exploits general properties of *read* and *write* operations.

The *knowledge* layer provides the means to enforce constraints which define software development policies. This layer may utilise (or not) the previous layer. Some approaches only provide syntactic constructs to model synchronisation, without reusing predefined strategies: synchronisation strategies must always be modelled from scratch. On the other hand, predefined strategies rest on some *consistency hypothesis*, and the knowledge layer must assume this hypothesis. For example, the 2PL protocol assumes that each transaction is an individual correct program: if it executes in isolation and starts its execution in a consistent state, then it terminates in a consistent state. In addition, knowledge is often used to break traditional isolation of transaction executions and to control dynamic interactions

Finally, the human *interface* layer assumes the interface between the process and human agents. It must allow human agents to create, control and terminate processes. It should also allow human agents to be aware of past, ongoing and future work. Transaction states represent valuable information for a human agent who observes and analyses a process. Finally, synchronisation specification can directly influence the process design process which is an important part of the interface layer.

6.3.1 Impact of the Repository on Consistency Maintenance

Among repository functionalities discussed in (Section 5.1.5, Chapter 5), we stress the following as crucial to support consistency maintenance.

6.3.1.1 Data Model

Maintaining consistency means enforcing constraints on object values and process states. The richer the data model, the richer the semantics which can be expressed by constraints. A consistency-maintenance mechanism based on an Entity-Relationship class model is naturally more powerful than a mechanism built on top of a file system. We particularly emphasise the use of process knowledge to relax serialisability of traditional transaction models as required by software processes.

New concepts introduced to enhance the modelling power of data models influence the complexity of transaction protocols. Conceptually, these new concepts reduce this complexity by filling the gap between the real word and its computer description. Technically, they increase it due to the need of complex techniques to manage complex objects (versioning, locking, copying, etc.). In addition, these techniques are not so mature.

An analysis of some of the most representative transaction models for PSEE, as described below in Section 6.4, indicates the importance of version management in transactions. It is not sufficient to distinguish between only two levels of consistency (consistent and inconsistent) as is traditional. It is necessary to distinguish between several levels of consistency. That is, at a given time, a *logical* object can exist in several different *version*s and these different versions must be maintained mutually consistent.

Another consistency-maintenance capability is the ability to provide different views of the object base to facilitate integration of the different points of view which different people can have of the process. The more integrated the process description, the easier this is to maintain.

6.3.1.2 Impedance Mismatch between Object Management and Operating Systems

The less the impedance mismatch between the operating system command language and the object management language, the easier it is to maintain consistency of software artifacts.

If tools do not operate directly on the internal process objects which represent the state of the current process, but on "converted" files, or through another additional interface, there is a risk of introducing inconsistencies.

In addition, we show below how the ability to store system concepts as persistent objects provides support for evolution and recovery and other interesting aspects, including the ability to model and integrate different synchronisation strategies[3].

6.3.1.3 ACID Transactions

There is always a level of abstraction at which a process is a serial execution of ACID short-term transactions. As a consequence, the repository must provide ACID transactions.

6.3.2 Workspaces: an Abstract Level to Support Flexibility

6.3.2.1 A Working Area as a Sphere of Isolation

A working area manager is intended to identify and to provide software developers with (only the) artifacts needed to reach their objectives. This helps avoid unintended, and unauthorised manipulation of objects, i.e, to preserve object integrity.

In traditional transaction models, a working area is seen as a sphere of isolation where a software developer can work individually. That is, when a transaction executes, it modifies copies of the objects which effectively persist in the object base. These copies define the working area of the transaction.

---

3. *Synchronisation strategy* is used as an alternative to *transaction protocol*.

However, strict isolation is not viable for software processes which are interactive 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 working areas, when they want. This idea is central to most transaction models for advanced applications.

This indicates that a working area basically supports the idea of transaction. That is, the working area manager provides capabilities to transfer object copies to a place where developers can work in isolation. This basic capability is extended to allow sharing of object modifications. This is specified by the definition of transfer operations between working areas, or rather by constraints on these transfer operations.

6.3.2.2 Reasoning on Transactions as Constraints on Transfer Operations

Relationships between transaction protocols and constraints on object transfers can be easily established. For example, the 2PL can be seen as the following transfer rules:

a) to operate on an object, a transaction must transfer it from the object base to its working area (*check out*),
b) an object which is shared between two working areas cannot be modified,
c) a transaction cannot transfer an object which has been modified from its working area to the object base (*check in*) before it commits,
d) a transaction which has read an object and frees this object for other transactions cannot check out another object.

Clearly, these rules imply completely isolated execution. Deleting rule *c)* is sufficient to break isolation. It is also sufficient to break all the guarantees of correct execution (of serialisability, i.e. correctness must be assumed in some other way). This shows that reasoning on consistency maintenance has much to do with reasoning on transfer operations between working areas, and vice-versa.

Returning to our example, this means that to make an object visible before its completion, a transaction must use other considerations to assert correctness of executions. Section 6.4 shows how more cooperative correctness criteria can be implemented in this way.

Another result demonstrated in [Unla92] is that, if an environment allows definition of a transaction protocol by integrating different (sub-)protocols, it is necessary to transfer i.e. to make a physical copy of an object which is checked-out, both for read and write accesses. Thus the working area manager must provide basic transfer operations which must not impose limits to transaction protocols, especially if we want to be able to allow different synchronisation strategies to run in different working areas.

6.3.2.3 Relationships between Working Areas

Relationships between process engines can be specialised to build a specific architecture: flat, hierarchical, multi-level, etc. Working areas can be similarly structured.

Typically, a process is often hierarchically organised. A process hierarchy reflects rich semantics about cooperation possibilities. Hierarchical organisation also fits the

idea of nested transaction. Thus, transfer operations are mostly related to parent-children relationships.

6.3.2.4 Persistent Working Areas

Workspaces will normally persist, i.e. working areas exist as objects of the object base. This can be used to maintain and express the relationships between working areas, transactions and processes. It can supply humans with query possibilities like:

a) which working areas contain a version of object *o?*

b) which is the version of the object *o* on which the process p operates?

c) which processes can access version *v* of object *o?*

Note that a persistent working area is close to a sub-object base and inherits some of the same qualities in case of failure, i.e the ability to recover a consistent state from a previous check point.

6.3.3 Predefined Synchronisation Strategies Layer

6.3.3.1 Classical ACID Transactions

The first assumption is that some synchronisation strategies can be defined independently of a particular software process. The second is that these strategies are sufficiently well defined and well accepted to be predefined. It is the fundamental idea of traditional database transaction model; it is the case of the classical pessimistic or optimistic transaction protocols.

As mentioned, traditional transaction protocols are too restrictive for software processes, due to the trivial operations they are supposed to support. Fortunately, predefined strategies do not imply a traditional strict correctness criteria such as serialisability and a rigid protocol such as ACID-transactions. These strategies must rather consider aspects of software processes such as user interaction, cooperation, long duration, uncertainty and so on. For example, in the COO system, a predefined protocol allowing intermediate result visibility has been defined. In Merlin, object consistency is maintained by synchronizing data accesses with four predefined classes of conflicts. Thus, the concurrency control protocol can allow interactive execution of processes.

6.3.3.2 Considering the Human Agents' Knowledge of the Process

In both COO and Merlin, resolving a conflict can ultimately rely on the human agent. In other words, the predefined strategies, whose goal is to synchronise processes without taking explicit account of process knowledge, exploit the knowledge of the human agents. This can be generalised: the transaction layer must also support the interaction between the environment and the human agents in a consistent way, at the appropriate place and time. This is especially interesting if we assume that a software process cannot be completely modelled in advance.

6.3.3.3 Programmable Concurrency Control

Such concurrency control implies that conflict resolution should consider not only (read/write) and (write/write) patterns. It should also exploit pre-programmed patterns of the processes involved.

6.3.3.4 Heterogeneous Protocols

Different agents in different working areas should be allowed to choose their own synchronisation strategy, their own concurrency protocol. However, these different individual strategies must be integrated into one global synchronisation strategy, a so-called *heterogeneous protocol*. This implies certain technical consequences, e.g. to copy an object each time it is reserved, independently of the mode with which it will be operated upon. There are also theoretical constraints on combinations of strategies, A PSEE transaction layer must provide not only predefined synchronisation strategies, but should also provide predefined means to combine them.

6.3.3.5 Persistent Transactions

We emphasise again that persistent transaction structures can facilitate reasoning about transactions, especially in the context of programmable concurrency control, of heterogeneous protocols, and to assist error recovery.

6.3.4 The Knowledge Management Layer

Knowledge management is crucial for consistency maintenance, whether the PSEE provides a transaction layer or not.

Note that predefined strategies to control concurrency were initially introduced for traditional database applications. For software engineering, synchronisation was generally explicitly programmed, from scratch.

The rationale behind each approach is very different. In the former domain, it is not possible to forecast all the kinds of interactions between processes; in the latter it is possible to completely specify processes and process interactions. In the former domain, we must define general, but restrictive synchronisation strategies. In the second domain, we must develop specific, but possibly more efficient strategies.

Both approaches exist in software engineering research, with current SEEs favouring either one approach or the other. Nevertheless, the trend is for a mixture of both approaches. It is not realistic to assume that everything can be forecast; it is also unacceptable not to exploit stated knowledge to relax the rigidity of predefined protocols.

6.3.4.1 From-scratch Synchronisation Strategies

Some systems do not provide a transaction layer but simply provide syntactic constructs to model synchronisation strategies.

The SPADE system provides a high level Petri Net formalism to model non traditional database applications, including complex dependencies among transactions. However, the synchronisation algorithm has to be specified from scratch.

The ADELE system fits very well with our architecture but with the transaction layer missing. Synchronisation strategies are programmable with powerful triggers which specify object sharing rules between working areas.

6.3.4.2 Predefined Synchronisation Strategy

As expressed above, any transaction model exploits knowledge of the process, even the basic 2PL assumes that serial executions are correct executions, i.e. that individual isolated processes respect constraints.

The same assumption applies for advanced transactions models.

In COO, it is assumed that pre-and postconditions of transfer operations respect some rules with regards to safety and liveness constraints. If not, execution correctness cannot be proved. More explicitly, it is possible to make visible an intermediate level only if the previously applied operations can be compensated in some way. Process knowledge is also used by human agents when they are asked to decide how to resolve a conflict. Process knowledge is also used in Merlin where a conflict resolution can depend on the processes which produced the conflict.

More generally, process knowledge is a key component in recovering a consistent state in case of failure or of reorganisation, i.e. to compensate a process instead of aborting it.

6.3.5 The Interface Layer

The interaction between human interface and consistency maintenance can be seen from four points of view:

a) the software developer who executes processes
b) the software developer or project manager who observes a process,
c) the software developer or project manager who makes some prediction on the next steps of a process,
d) the process designer.

When executing a process, software developers are directly influenced by the conflicts which occur. In case of automatic synchronisation, the interface must at least report on conflicts and their consequences. In the case of interactive conflict solving, the developer contributes directly to synchronisation, deciding on whether or not to allow shared writing, or simultaneous reading and writing of an object.

When a developer observes a process, the transaction structures contain information on the current and past operations which have been applied to objects, and by which processes. For instance, we may request which other developer concurrently accesses a given object.

Prediction of process future and consistency maintenance are related. In most cases, both transactions protocols and planning mechanisms exploit the same process knowledge, i.e. integrity (safety and liveness) constraints. Transaction structures can also be directly exploited for this purpose. As an example, inter-dependencies among pre-declared transaction intentions are directly exploited to support planning and impact analysis.

Finally, (sub)-process synchronisation must be considered by the process design process: different synchronisation rules can lead to different process styles. In COO, for instance, processes may execute in a serialisable way or in a cooperative way. A first set of design rules produces process models which allow only serialisable executions. A second set of design rules produces process models which allow cooperative executions.

6.4 Current Work

6.4.1 The COO System

The COO system is a research prototype being developed by the ECOO team at CRIN. Its current version executes in a PCTE context. A new version is being designed to support cooperation in a Java - Internet Environment with the objective of addressing a larger set of applications. The COO project is a continuation of the ALF project to further research the problems related to cooperation support and especially maintaining consistency between software artifacts.

6.4.1.1 Organisation of Processes

A COO process breaks down into sub-processes. Each (sub)process executes in its own working area governed by its knowledge. Each developer operates through their own interface upon objects in their working area: a sub-database which consists of the cooperative versions of the objects in the common repository.

6.4.1.2 The *Repository* Layer

The COO repository, called P-RooT[4], implements an object oriented version of the PCTE[5] interfaces. Its data model is based on an ERA [Chen76] data model extended with inheritance, and different categories of relationships between objects (composition, reference, designation, existence). The *PCTE* data types are defined in schemas which are used to define the view a process has of the object base. In fact, each process is associated with a list of schemas, called its *working schema*. The view of the process on the object base is constrained by the union[6] of the type definitions of each schema

4. P-RooT stands for PCTE Redesign with Object Orientation Technology
5. Portable Common Tool Environment
6. Union of two schemas means, for each object type, the union of its properties. Problems due to synonyms and homonyms are directly resolved by the typing system

in its working schema. Processes are instantiated in the object base as objects. Thus, PCTE provides an integration of both operating system services and database system services. PCTE provides a closed nested transaction mechanism based on a locking mechanism. The P-RooT trigger mechanism is based on a simple event-action mode where an event is an operation call and an action is an operation invocation.

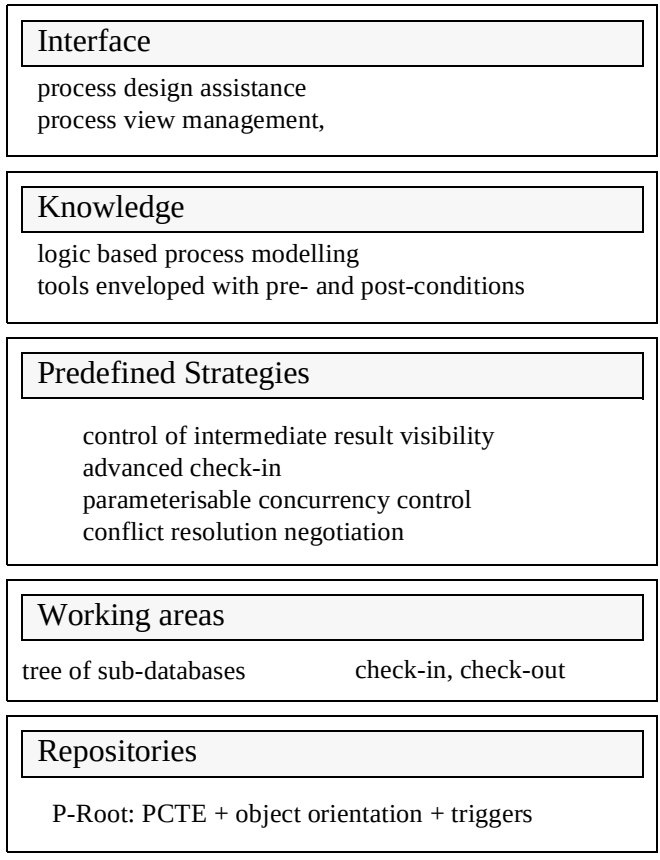

Figure 6.4 COO Architecture

6.4.1.3 The *Working Areas* Layer

The working area layer implements an object base/sub-base architecture. It provides object identification services and version management services. The working area layer organises the object base into a hierarchy of working areas. Thus, each working area has a parent working area except the initial one which is the root of the working area hierarchy and which directly feeds into the *repository* object base.

To operate on an object, a process must transfer it from its chain of ancestor working areas to its own working area: this transfer is done by the *check out* operation. This operation creates a new version of the object which is inserted in the working area. Thus, several physical copies of the same reference object can exist in different working areas. The physical copies of the reference object, we call a reference object a *variant*, are related together through a version graph, i.e. successor/predecessor links. Versioning is transparent to processes and a process issues a request to a variant object. To identify the version the process must operate on, a mapping is built which associates a physical copy of the variant object with the working area from which the request is issued. One can transfer a copy of a variant object in the working area into its parent working area: this is done by the *check in* operation. After a *check in* operation, the transferred copy of the variant replaces the previous value of this variant in the parent working area. The *update* operation is used to (re)synchronise an object in a working area with the current version of this object in its parent working area. This creates a new version in the working area of the process which initiated the request. This new version has two predecessors: the (old) version in its working area and the version in its parent working area and is built by *merging* the two ancestor versions. The *merge* operation is done in the current working area. Due to the variety of merging operations, we do not provide a general *merging* operation; nevertheless, we will discuss a special case of merging in the next section.

Workspaces persist as PCTE objects. This is of interest in the case of a logical or physical failure. It means also that a working area can be considered as *a sub-database*.

6.4.1.4 The *Predefined Strategies* Layer

In COO, we distinguish between three levels of consistency:

a) globally stable objects.
b) locally stable objects.
c) intermediate result objects.

Globally stable objects are objects which are consistent with regards to any process definitions. Locally stable objects are objects which are consistent with regards to the local process definition, but which can be inconsistent with regard to one or several enclosing processes. Intermediate results objects are objects which can be inconsistent with regard to the definitions of the process which produced them, and which can be operated on again by this process.

Processes are hierarchically organised and when a process wants to access an object, the version of the object it will attempt to obtain is: the closest intermediate result of this object if it exists, or the closest locally stable object if it exists and no intermediate result exists, or finally the globally stable object.

Clearly, visibility of intermediate results allows the relaxation of the constraint on isolated execution by allowing a process to see intermediate results of siblings. Intermediate results are transferred by a transfer activity called *upward commit*. To maintain consistency with intermediate results, we say that, when a process *A* has read an intermediate result of a process *B*, *A* depends on *B*. When a process depends on another proc-

ess, it cannot *terminate* without *updating* the intermediate result in the dependence with the corresponding final value. The *refresh* operation is a special case of *update* where the *merge* operation consists simply in the replacing of the value in the child working area by the value in the parent working area. By nature, commit dependencies can be crossed, and more generally, the dependency graph can be cyclic. In such a case, all processes in a cycle are requested to terminate "simultaneously" with the same values of objects involved in the dependencies. This is implemented by a kind of *two phase commit* protocol inspired by [Gray78]. *Check in* is now reserved for the transfer of final results (marked as final). It means the objects cannot be modified again by the same process. We authorise (advanced) *check in* of an object.

A COO process executes as a nested transaction. The root transaction represents the whole process, at the leaves are atomic processes called activities, at the intermediate levels are compound processes called tasks. A task is simply a synchronisation entity: it delegates object modifications to its enclosed activities. Each time a process is initiated, structures are created, including a new working area and a *check out* is done on all input (parameter) objects. A transaction *completes* when the enclosed process succeeds in executing its *terminate* activity. The effect is to *check in* any results which have not already been *checked in*.

An intermediate result can be produced only if the process which produces it is compensatable. In other words, transfer operations must be constrained, and, in the worse case, intermediate results must be prohibited. To support this unfavourable case, the default protocol implemented in COO is the 2PL nested protocol, where *check out* and *check in* encapsulate respectively the role of *read* and *write*. Atomicity of basic activities is assumed by the repository level: all activities, including transfer activities, execute in isolation. Then from this basis, to support interactions, the 2PL compatibility table can be modified to allow, on the authority of human agents, a transaction to *upward commit* an intermediate result, i.e. a transaction to read an intermediate result of another transaction and two transactions to write the same object. This decision rests on the responsibility of the human agent or can be programmed depending on the processes in the conflict.

Finally, note that all transaction structures persist. This is of the greatest importance: in recovering a consistent state in the case of a logical or physical failure, in defining new predefined synchronisation strategies, and in integrating different strategies in the same protocol.

6.4.1.5 The *Knowledge* Layer

The transaction layer assumes consistency on the basis that processes are consistent with respect to their definitions. As the COO PML is logic based, activities, including transfer activities (check in, check out and upward commit), are extended with pre- and post-conditions.

6.4.1.6 The *Interface* Layer

COO provides a means to assist process design. We distinguish between a conceptual level where software development rules are expressed in a limited form of temporal logic and a logical level where these rules are implemented in terms of pre- and post-conditions on activities.

The behaviour of a process is defined by aggregating several schemas which define its context. It means at least the P-RooT schema, the Workspace schema, one Transaction schema, and one Constraint schema. This provides a very flexible and extensible way to model and enact processes. As an example, for a given process model, choosing between a serialisable and a cooperative execution mode is simply a matter of choosing between one constraint schema and another.

6.4.1.7 The Cooperation Example

As a short illustration, let us consider how COO manages the situation depicted in our motivating example in Section 6.1.2, Scenario 2.

Table 6.5 A correct scenario

POCM(A)	POCM(B)	POCM for Produce the object code for a module
init(POCM(A))*	init(POCM(B))*	
edit_interface(A)		
		POCM(A) upward commits the intermediate value of the interface of A (when edit_interface(A) completes)
upward_commit(interface(A))		
	edit_interface(B)	
		POCM(B) checks out the interface of A, intermediate result of POCM(A)
	check_out(intface(A))	
	edit_body(B)	
	compile_body(B)	
edit_body(A)		
compile_body(A)		
edit_interface(A)		
		POCM(A) upward commits the new value of the interface of A
upward_commit(interface(A))		
edit_body(A)		
compile_body(A)		
edit_interface(A)		
upward_commit(interface(A))		

Table 6.5 A correct scenario

	refresh(interface(A))	**POCM(B) refreshes its current value with this new value**
	edit_body(B)	
edit_body(A)		
compile_body(A)		
terminate(POCM(A))*		**POCM(A) declares all its results as final results, including the interface of A**
	compile_body(A)	
	terminate(POCM(B))*	**as it has read the final value of the interface of A, POCM(B) can commit**

* *init* and *terminate* are COO activities introduced to manage pre- and post-conditions of compound processes. Especially, when a process terminates, it checks_in all its results which are outstanding. As a consequence, it declares them as final results.

Circulation of the interface of A between workspaces in the related scenario

Figure 6.5 shows the circulation of the interface of A between POCM(A) and POCM(B). This is controlled by Coding Task. In particular, Coding Task will not accept POCM(B) to terminate if it has not read the final value of the interface of A.

Process Modelling

A process model (PM) in COO is a 7-tuple < S, V, H, P, O, I, C> where

a) S is the *Signature* (parameter types) of the PM,
b) V is the *View* of the PM on the object base,
c) H is the *Human* Agents (roles) needed to enact a process of the PM,
d) P is the *Precondition* of the PM,
e) O is the *Objective* of the PM,
f) I is the *Implementation* of the PM, i.e. the list of sub-process models,
g) C a set of *integration constraint*s which describes how sub-processes can interact to reach the goal of a process they implement (we distinguish between safety and liveness constraints). For more about this model, see [Goda93a, Goda95].

The process Coding Task which governs our example scenario is primarily related to the liveness constraint which says “if the interface of A is modified and B is a module which depends on A, the body of B must be modified before Coding Task terminates”.

In our PML, it is translated:

from *new((interface(A)) and depends_on(A,B))* ***sometime*** *new(body(B))* ***before*** *terminated*

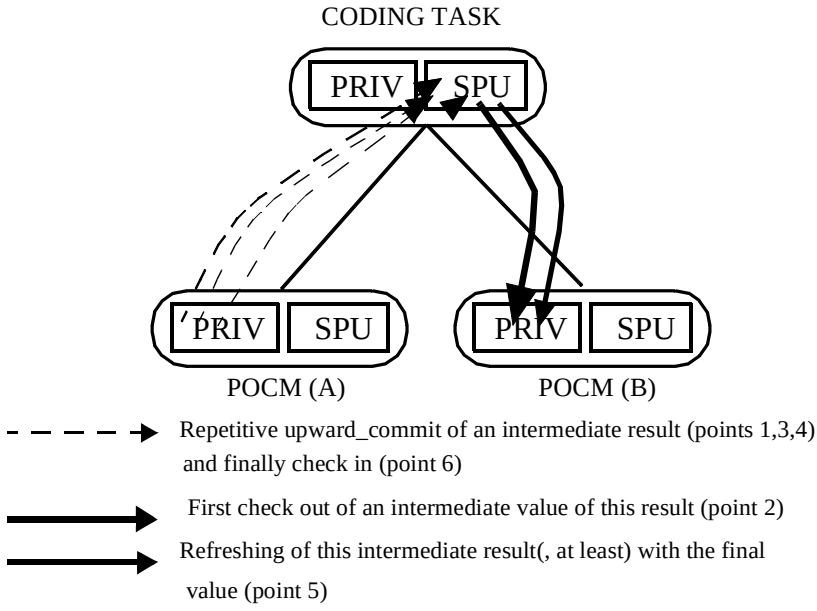

Figure 6.5 Circulation of the interface of A**.**

To maintain this constraint, we introduce the predicate: inevitable_before (I_B(A_i,B)) which allows the storage of the useful history. If this predicate exists, it means that a value of the interface of A (A_i) needs to be consumed to update the value of the body of B (B_b) before Coding Task is able to terminate.

We can transform this constraint into preconditions and postconditions of the activities as shown in the following table. This table results from the application of our transformation rules as described in [Goda95].

Activities in POCM(A)	**Activities in POCM(B)**	**Preconditions**	**Postconditions**
upward_commit(A_i)			insert[a](I_B(A_i,B)) with B depends on A_i
checkin(A_i)			insert(I_B(Ai,B)) with B depends on Ai
	checkin(B_b)		delete(I_B(A_i,B))
	terminate	there is no I_B(A_i,B)	

a. “insert a tuple” effectively inserts this tuple only if it does not already exist.

6.4.2 The MERLIN System

The MERLIN system has been developed by the Software Engineering team at Paderborn with a particular interest in the impact of cooperation on consistency maintenance and user interfaces.

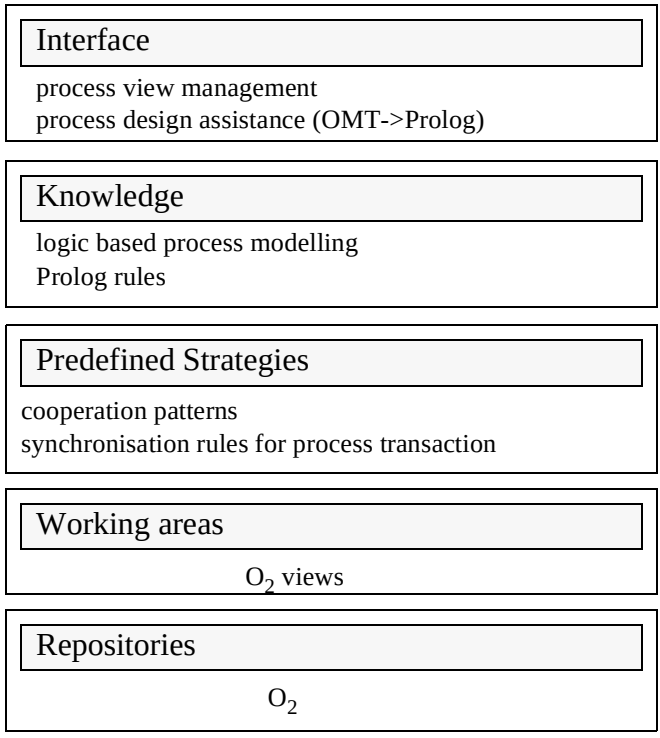

Figure 6.6 MERLIN Architecture

6.4.2.1 The *Working areas* Layer

In contrast with many other PSEEs the Merlin Session Layer is not implemented as a base/subbase implementation with check out and check in functionality. All documents of a particular configuration reside in a common repository. For each user the Workspace Management (cf. Section 5.1.4, Chapter 5) creates his/her personal workspace as a view on all documents depending on the users identity, role and access rights. In this approach intermediate modifications are immediately propagated to all other users with changed documents in their workspace.

Transactions are initiated by a user corresponding to his/her personal workspace, e.g. the user may start a working context transaction (wcCP)[7] to control all activities performed on documents in this workspace. In the case of conflicting transactions, such

that an activity (e.g. editing a document) has to be aborted, the user can store his/her intermediate results as a new document version that may be merged with the current version after the other conflicting transaction has committed.

6.4.2.2 Cooperation Patterns

Cooperation Patterns (CP) have been developed in Merlin to coordinate multiple users' work. They correspond to transaction types. In "traditional" applications only one type of transaction exists, namely the ACID-like transactions mentioned in Section 6.2.1 Merlin distinguishes CPs used to control user performed activities and CPs used to control activities which are performed automatically by the PSEE.

The CPs used to control user performed activities differ with respect to the number of objects they access. A CP either controls the execution of a single activity on a single document (e.g. editing a module) or it includes and controls all activities performed within a working context (e.g. editing several source code modules, compiling and linking them). The first one is called activity CP (*aCP*) and the second one working context CP (*wcCP*).

Merlin further distinguishes two CPs to control automatically executed activities. Either the PSEE follows some automation conditions in the process definition, and therefore changes the states of documents and invokes tools (e.g. compilation of a module after the imported modules have been finished), or document accesses and corresponding document state changes are triggered by some consistency conditions (e.g. recompilation of an already compiled module because an imported module has been modified). The CPs corresponding to the latter two types of activities are called automation CP (*autoCP*) and consistency CP (*consCP*) respectively (this distinction is similar to the separation between automation chains and consistency chains in Marvel [Barg92]).

Each transaction, or rather activity performed, is considered to be an instance of one of the above defined CPs. As those transactions are based on specific knowledge about a software process model definition, we call them process transactions in order to distinguish them from ACID-like transactions in the traditional sense. A process transaction is defined as a tuple **T = (i, cp, u, r, L, T, p)** with

a) i:the transaction's **Identifier**

b) cp:**CP** applied for the transaction cp ∈ {*aCp*, *wcCP*, *consCP*, *autoCP*}

c) u: Identifier of the **user** who initiated the transaction

d) r: **role** of the user who initiated the transaction

e) L: **Locks**, $L \subseteq \{l_1, ..., l_n | \ l_i \text{ is lock}\}$ with $l_i \in$ {w(D),r(D), w(state(D)), r(state(D)) | D is a Document}

f) T:**Timestamps** with $T \subseteq \{t_1, ..., t_n | \ t_i \text{ is timestamp}\}$ with $t_i \in$ {tw(D,T), tr(D,T), tw(state(D),T), tr(state(D),T)| D is a document, T is a point in time}

g) p: Identifier of the **parent** transaction

---

7. Cf. the earlier definition of process transaction and cooperation patterns.

Besides a unique identifier of a process transaction and the CP which defines the synchronisation of the transaction with others (see below), the user who initiated the transaction as well as the user's role are associated with the transaction. This information is provided to users involved in a concurrency conflict.

Transactions are synchronised either in a pessimistic or optimistic way. In the pessimistic case attribute *Locks* describes the locks held by a transaction. In the optimistic case attribute *Timestamps* describes a set of timestamps. The last attribute *parent* contains the parent transaction identifier, if the transaction is initiated as a subtransaction.

6.4.2.3 Synchronisation Rules for Process Transactions

The following three sets of rules form the fundamental basis for synchronizing concurrently running process transactions.

(1) All transactions except those of type *aCP* are synchronised in a pessimistic way, i.e. if a transaction is started all needed locks are acquired. The transaction is not started if a lock cannot be acquired. A transaction of type *aCP* can run in an optimistic or pessimistic mode. In the case of an optimistic mode, its timestamps are validated at the end of its execution. If another transaction has acquired locks the optimistic transaction of type *aCP* is aborted. If another optimistic transaction of type *aCP* has committed already and has accessed the same objects while they were accessed by the transaction to be validated, the transaction is also aborted.

(2) Synchronisation is based on the definition of priorities. In case of a conflict the access right is usually granted to the transaction that has the highest priority whereas the other is aborted (exceptions from this rule exist, as the example below will show). Transactions of type *consCP* have the highest priority because they are applied to preserve the consistency of a process. Next at the same level of priority are transactions of type *wcCP* or *aCP*, if the pessimistic mode has been chosen for the latter one. Next are transactions of type *autoCP*. Transactions of type *aCP* have the lowest priority, if the optimistic mode has been chosen.

(3) Transactions of type *consCP* or *autoCP* are always executed as subtransactions of transactions of type *wcCP* or *aCP*, because activities automatically executed by the PSEE are always triggered by a user activity. The subtransactions have access to the locks still held by the parent transaction. In case the parent transaction is of type *aCP* in optimistic mode the timestamps are transformed to locks at validation time. If the validation fails, the parent transaction is aborted.

These rules avoid the situation where *consistency* or *automation* activities triggered by a user's activity cannot be performed because in the meantime locks were acquired by other transactions. For example, a compile activity is triggered and performed when a user has finished editing a module. Consequently, a project, i.e. all its corresponding documents, is always in a consistent state.

Based on these rules we have defined a number of more sophisticated synchronisation rules which we do not give in full detail here but refer to [Junk94]. The following example should illustrate that even withdrawing locks is a possible conflict resolution.

Consider the conflict between a transaction of type *wcCP* or *aCP* (in optimistic mode) on one hand and a transaction of type *consCP* on the other. In such a conflict, it is distinguished whether or not the *wcCP*- or *aCP*-transaction has already started subtransactions.

The first case means that the user has already finished his activity in the working context. If a subtransaction (of the *wcCP* or *aCP* transaction) of type *consCP* concurrently accesses the parent's lock which causes the conflict, the requesting *consCP* is aborted. If a subtransaction of type *autoCP* currently accesses the parent's locks, the transaction of type *autoCP* is aborted and the lock is withdrawn from the parent transaction of type *wcCP* or *aCP*. If no subtransaction currently accesses the parent's lock, the lock is withdrawn from the transaction of type *wcCP* or *aCP*.

In the second case (no subtransactions initiated yet, i.e. the user activity is still running) it has to be distinguished whether the conflict is caused by concurrent accesses to a document or a document state. If the transaction of type *consCP* requests a state lock, this lock is divested from the pessimistic transaction of type *wcCP* or *aCP* without aborting the transaction. This solution is based on the specific application knowledge that a document's state is only changed by the user at the very end of the development activity. Withdrawing the state lock enables the user to continue the activity (e.g. editing a source code module), but does not allow him to change the module's state at the end of his activity. In order to change the state, the user has to finish his/her activity and to start a new one later. Note, that the performed modifications are not lost. Only if the conflict between *a consCP*-transaction and a *wcCP*- or *aCP*-transaction concerns a document's lock, the transaction of type *wcCP* or *aCP* is aborted. Then the already performed modifications are stored in the user's private workspace. Thus a new version of a document is created. Furthermore the "losing" user is informed about the user who caused the conflict (and that user's role) such that the two users are able to discuss further synchronisation of their work, possibly off-line.

6.4.2.4 The *Knowledge* Layer

The Merlin transaction model and in particular the synchronisation rules are formally defined in a set of PROLOG rules. These rules are efficiently executable and act as an inference engine for software processes. A particular software process is defined in an OMT-like notation [Junk94] which is mapped to PROLOG facts and thus can be interpreted by the inference engine.

Further PROLOG rules define all the preconditions which have to be fulfilled in order to build a particular user's working context or to execute a particular activity. A particular working context or the intended execution of an activity are described as PROLOG goals. By interpreting the PROLOG program the PSEE checks preconditions, for example, a user's responsibilities, access rights or document states. An activity whose preconditions are true is displayed in a working context (possibly on demand) or automatically executed (e.g. compile). Performing an activity could result in new preconditions becoming true (based on the user input of status information, or on the results of automatic activity executions) and consequently new activities displayed. The

PSEE's major job in the context of managing working contexts is thus to evaluate all preconditions and to refresh all displayed working contexts accordingly (on demand).

6.4.2.5 The *Repository* Layer

In Merlin the fully object oriented database system, O_2 [Banc92] is used to maintain all information about software development projects including access rights on documents and document states. The schema is defined using O_2C which is an object oriented extension of the programming language C. Furthermore O_2 provides an implementation of the standard DDL/DML for object management systems proposed by ODMG [ODMG97]. O_2 provides a simple ACID transaction mechanism for optimistic and pessimistic transactions. The Merlin transaction model uses pessimistic (short time) ACID transactions to modify those PROLOG facts that reflect the current state of the cooperation model.

6.4.2.6 The Example Scenario

In this section we apply Merlin process transactions to the sample scenario. Table 6.6 maps each activity of the scenario (middle column) onto the corresponding process transaction[8] (right column). Nested process transactions are explicitly marked by a shift to the right. Each event (starting or terminating process transaction) that results in a modification of a document state is given a unique number in the left column. These event numbers are used to illustrate the transitions of the four documents in Figure 6.7 over.

Table 6.6 Application of Process Transactions to the Scenario

Event	Activity	Process Transaction
1 1.1	edit_interface(A)	(i_1, *aCP*, -, T_1) (i_2, *autoCP*, {w(state(body(A))}, i_1)
2 2.1	edit_interface(B)	(i_3, *a*CP, -, T_2) (i_4, *autoCP*, {w(state(body(B))}, i_3)
3	edit_body(B)	(i_5, *aCP*, -, T_3)
4	compile_body(B)	(i_6,*autoCP*, {w(state(body(B))}, i_5)
5	edit_body(A)	(i_7, *aCP*, -, T_4)
6	compile_body(A)	(i_8, *autoCP*, {w(state(body(A))}, i_7)

8. In this notation of process transactions identity and role of users are skipped for sake of better readability, since there is no need of these attributes here.

Table 6.6 Application of Process Transactions to the Scenario

Event	Activity	Process Transaction
7 7.1 7.2 7.3 7.4	edit_interface(A)	(i_9, *aCP*, -, T_5) (i_{10}, *consCP*, {w(state(interface(A))}, i_9) (i_{11}, *consCP*, {w(state(body(A)), w(state(body(B)))}, i_9) edit completed (i_{12}, *autoCP*, {w(state(body(A)), w(state(body(B)))}, i_9)
8	edit_body(A)	(i_{12}, *aCP*, -, T_6)
9	compile_body(A)	(i_{13}, *autoCP*, {w(state(body(A))}, i_{12})
10 10.1 10.2 10.3 10.4	edit_interface(A)	(i_{14}, *aCP*, -, T_7) (i_{15}, *consCP*, {w(state(interface(A))}, i_{14}) (i_{16}, *consCP*, {w(state(body(A)), w(state(body(B)))}, i_{14}) edit completed (i_{17}, *autoCP*, {w(state(body(A)), w(state(body(B)))}, i_{14})
11	edit_body(B)	(i_{18}, *aCP*, -, T_8)
12	edit_body(A)	(i_{19}, *aCP*, -, T_9)
13	compile_body(A)	(i_{20}, *autoCP*, {w(state(body(A))}, i_{19})
14	compile_body(B)	(i_{21}, *autoCP*, {w(state(body(B))}, i_{18})

Figure 6.7 is a diagram that shows the sequence of state transitions for each document in the example. States are represented by ovals while directed edges denote transitions between states. The initial state is as noted at the top of each diagram. Edges are annotated with event numbers in order to have a mapping with process transactions that cause the corresponding transitions (Table 6.6).

6.4.3 The ADELE System

The ADELE system is developed at the LSR-IMAG laboratory in Grenoble. It was initially designed to manage versions and configurations during software development, and there is a commercial version that is confined to this task. In this section we will

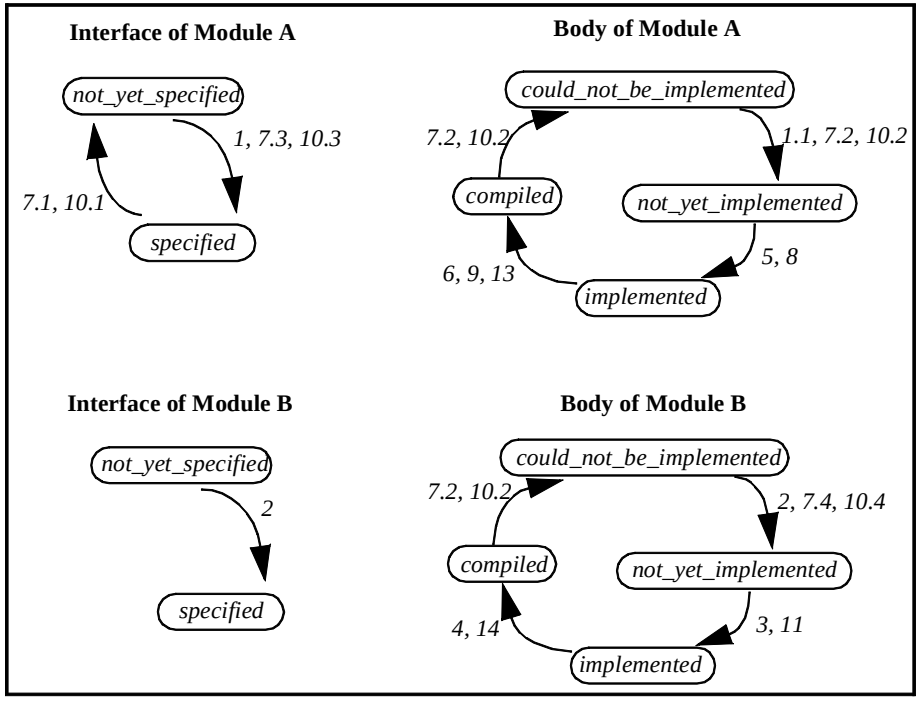

Figure 6.7 State Transitions of Evolved Documents

consider an enhanced version, which the objective of improved support for “Team Work”.

6.4.3.1 The *Repository* layer

ADELE proposes its own data repository as default. The ADELE data model is designed for the support of Software Engineering work in general, and configuration management in particular. It features an EER (Extended Entity Relationship) model, where relationships are first class citizens, with identity, attributes and methods, in the same way as objects. Multiple inheritance is available for both objects and relationships. Special attention was devoted to complex object control. Composition relationship semantics is not predefined by the system but explicitly declared by the data modeller. Components of complex objects can be automatically found using explicit built rules. This is an extension of configuration construction, where components can be found using the “dependency” relationship and selection rules.

ADELE considers the File Systems as one of its repositories, and as such, most commands can work in the same way on files in a file system or on its data repository. Recovery and transactions span over all the repositories.

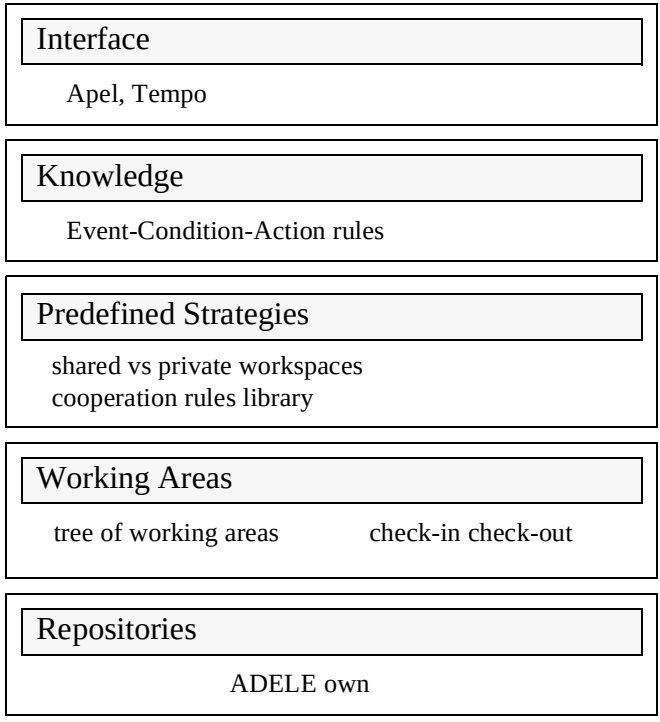

Figure 6.8 ADELE Architecture

6.4.3.2 The *Working Areas* Layer

A working area, called Workspace (WS), is a kernel concept. It is built as a part of the DB. A WS is the implementation of a sub DataBase: an identified sub set of another WS, where the original WS is the repository. Thus WSs are nested, the DB being a tree of WSs.

The kernel WS manager recognises two classes of WSs, Database WSs and File System WSs. A Database WS is a sub-DB, i.e. a sub set of the entities and relationships pertaining to another WS. The WS manager is responsible for the identification of the entities, to ensure transfers between the original WS (the parent), to the destination (the child), and to avoid any malicious or unintended access to objects not pertaining to the WS. All transfers are implicit and transparent.

WSs raise the granularity of most commands from a single file to the whole WS, for example: create, delete, resynchronise. WSs are created by the make Workspace command “mkws object-list -t WS_type ...”, which is issued from parent WS, by default the repository (i.e. the root WS). This granularity is much closer to the engineers’ conceptual world. For instance “mkws V2Conf -t develop -u Jim” creates for *Jim* a *develop-*

ment WS containing the configuration *V2Conf*, itself containing perhaps thousands of files (in other words, files can be checked-out set by set).

A File System WS is the file system image of a DB WS i.e. all WS object representing a file is really a file in a file System. A File System WS acts as an extended file system, with attributes, relationships and abstract objects. A component, the "File System Mapper" is in charge to translate any command argument from files into their corresponding objects. ADELE File System WSs are *transparent*, i.e. users and tools work as normally in a real file system, with no changes in routine and (almost) no overhead.

6.4.3.3 Predefined Strategies

Shared or private workspaces

A WS has a parent (except the root WS), and may be one of several different children. Each child contains a sub set of the objects of its parent. *Cooperation* refers here to the way the "same" object is managed in both WSs. The kernel WS manager recognises only two cooperation policies: Shared and Private.

Shared means that the parent and its child really share the same object, i.e. a change is immediately visible in both WSs. It exists physically as a single object, with no concurrency control except the DB short ACID transactions, and the explicit locks provided as part of the Configuration Manager.

Private, conversely, means a WS acts as a (very long) ACID transaction containing persistent objects i.e. changes are local to the WS, and any change performed outside the WS are not visible. In long transactions, as soon as an object is modified, the kernel WS manager creates, transparently, a copy of that object, visible and accessible only in that WS. In short transactions, a lock is set to prevent conflicts. At WS commit changed objects are resynchronised (i.e. merged) with their original in the parent WS. Change propagation is only allowed along the WS tree, and at WS completion.

WSs are typed, and a different policy can be set for arbitrary set of objects. Thus a part of a WS can be shared, and the rest can be private. Using only the kernel WS manager, most of the usual cooperation policies can be implemented.

Rules library

An advanced feature of ADELE is that it allows tried and tested synchronisation strategies to be organised in libraries. In other words, a strategy which has been developed in a process, and whose correctness has been established through "experimentation" can be stored to be reused in another process. Strategies are expressed as Event-Condition-Action rules, see Section 6.4.3.4 .

6.4.3.4 The *Knowledge* Layer

A strategy formally defines which kind of data must be transferred between WSs, at which time, and between which WSs. It extends the kernel WS manager in that (1) cooperation can be implemented between any pair of WSs with common objects (not only the parent/child tree), (2) the data exchange can take place at any defined instant

(not only at WS commit), (3) different policies can be active simultaneously for a given object with respect to different coordinated WSs (instead of a single parent coordinated WS), and (4) cooperation policies can be set, removed and modified at any time between any WS pair.

The DB becomes a network of coordinated WSs, exchanging information in a predefined manner, and following an explicit model.

A WS type is defined explicitly, WS instances are objects themselves. The process engine notifies the WS manager each time an event occurs on an object. The WS manager, if the object is coordinated, and depending on the coordination policy(ies), discards the event or executes the action defined in the policy. (The action may be merge both objects, notify the user, ask the user what to do etc.).

A WS type is defined as follows: WStype = (Content*, Sub WSs*, coordinated*, role) with

a) *content*, the list of object types contained in the WS,

b) *Sub WS* the child WS types,

c) *Coordinated*: Coord = {OrigWS, DestWS, which, when, policy}, with *OrigWS* and *DestWS* the coordinated WS pair; *which* a filter defining which objects are the subject of the policies, *when* is a condition expressing when the policy must be executed, and *policy* is what is to be done when the *when* condition becomes true. Examples of coordinations are given in C1-C4 below.

Integration = (SourceCode, {Analysis, Development}, {C1, C2, C3, C4}, Integrator) where:
C1 = (Self, Analysis, SourceCode, True, shared)
C2 = (Self, Development, SourceCode, True, private)
C3 = (Development, Development, (!type = Header), save, resynch),
C4 = (Development, Development, (!type = Body), become_ready, notify);

Line 1 expresses that *Integration* WSs contain *SourceCode*, have sub WSs *Analysis* and *development*, uses coordination C1 to C4, and are used by engineers with the role of *Integrator*.
C1 expresses that *Analysis* WSs are *sharing* the source code with the Integration WS (*self*). C3 expresses that between any two *Development* child WSs, the *headers* are to be *resynchronised* as soon as a modified *Header* is *saved*.
C4 expresses that a notification (*notify*) is to be sent to the WS responsible as soon as a program *body becomes_ready*. The event *becomes_ready* being defined as “event becomes_ready = (!cmd = mda and !a = state and !val = ready)” i.e. the state attribute is changed to ready.

C1 and C2 simply use the kernel WS facilities, while C3 and C4 define advanced cooperation policies, based on a library of predefined policies.

6.4.3.5 The *Interface* Layer

In particular, this layer supports the modelling of software project activities. It is organised in two parts. The basic process engine language, based on triggers, and two advanced formalisms: the Tempo and APEL process languages for the definition and enactment of process models.

The basic process engine interprets all the events occurring in the system according to the current process model. Events are all accesses to the system, or accesses to any data (objects as well as relationships) contained in any repository. Pertinent events are explicitly declared. Triggers are Event-Condition-Action rules, and are defined as part of a type definition.

The high level languages include constructs to describe: (1) the *activities* related to a life cycle model, (the model specifies the objects to manipulate, tools to invoke, the authorised agents, and constraints to check the validity of operations carried out in an activity), (2) *relationships* between the activities, (decomposition of an activity into subactivities) (3) the *coordination* among the activities (sequencing, parallelism, data flow) which specifies certain constraints on the execution of activities.

Tempo is a textual language emphasizing an object oriented style for process description. One feature of Tempo related to synchronisation is that it offers advanced mechanisms for cooperation support. We have identified two major capabilities. The first is the process layer which defines specific scenarios for cooperation and synchronisation appropriate for these scenarios. The software process is specified as long term and complex events which perform activities in response to appropriate conditions. The second is the cooperation layer which controls the degree of parallelism in resource sharing allowing different strategies on a per-application basis, without using transactions hierarchies.

In combination with Tempo, APEL (Abstract Process Engine Language) is a graphical language primarily designed for capturing and understanding processes. Then, by recursive refinement, the degree of description becomes sufficient for generating the code for enaction. APEL offers process engineers four aspects: Control flow (including multi instance activities, control by event), data flow (including asynchronous data transfers), data definition (including versioning and measure aspects), and Work Space (including cooperation policies, user role and State Transition aspects).

6.4.4 The SPADE System

The SPADE system [Band93, Band94, Band94b] is developed by the Software Engineering Team at Politechnico di Milano. Software processes in SPADE are specified using a high-level Petri net notation which covers all aspects of processes, including cooperation.

6.4.4.1 SPADE Architecture

The SPADE architecture is structured in four different layers: the Repository, the Process Engine, the User Interaction Environment, and the SPADE Communication Interface. The Repository stores both process model fragments and process artifacts. It has been implemented on top of the O_2 database. The Process Engine is responsible for the execution of the SPADE LANGuage (SLANG) process models. Different fragments of a process are associated with one SLANG interpreter that executes as a thread of the Process Engine. The User Interaction Environment (UIE) is a collection of tools, available in the development environment, that take part in the process enactment. Process users always interact with the Process Engine through the services provided by these tools. The SPADE communication interface (SCI) behaves as a communication bus, connecting SLANG interpreters with tools in the UIE. The SCI defines a message based control integration protocol. To extend the scope of the process model control to other tools that do not know this protocol, a number of bridges can be added to the SCI. A bridge is a gateway that translates control messages from the SCI protocol into another control integration protocol. SPADE comes with bridges for DEC FUSE, SUN ToolTalk and Microsoft OLE/DDE.

6.4.4.2 SPADE Language

SPADE defines a process modeling language called SLANG that supports process description from two different perspectives. The data perspective is given by a set of classes (or abstract data types) called *ProcessTypes*, and the workflow perspective is given as a set of activities called *ProcessActivities*. The two process views are integrated: the data manipulated by activities in *ProcessActivities* are instances of the classes contained in *ProcessTypes*.

The *ProcessTypes* set is organised in a class generalisation hierarchy, following an object-oriented style. Each class has a unique name, a type representation, and a list of operations, that may be applied to objects of the class. Each class can be used to represent different kinds of process data in the process model. Process data includes: process products and by-products (source code, executable code, design documents, specifications, etc.), resources and organisational process aspects (such as roles, skills, human resources, computer resources, etc.), as well as process model and state information (for example, activity definitions, activity instances, class definitions etc.).

Each activity in the *Process Activities* set describes the workflow within a process fragment. Activities are specified using a high-level Petri net notation. Petri net transitions represent process steps and Petri net places behave as typed data (token) containers. The Petri net topology of an activity describes precedence relations, conflicts, and parallelism among process steps. Usually, a transition affects only the data contained in its pre-set and in its post-set and corresponds to a transaction in the underlying database. Some special transitions, called *black transitions*, are able to affect the User Interaction Environment by executing tools and invoking the services provided by these tools. An activity definition may also include the invocation of other activities.

6.4.4.3 The Concurrent Coding Activity Example

To show how coordination and data sharing are managed in SPADE, let us consider the motivating example presented earlier Section 6.1.2 . This process is the parallel coding of two modules, A and B. These two modules are interdependent. In particular, B depends on A. According to the Scenario 2 (see Table 6.2), the coding of the modules can be performed in parallel, provided that the last version of module B is generated using the last delivered version of module A.

Figure 6.9 shows the SLANG activity that describes the process of coding one of these modules. The activity is parametric with respect to the module to be developed, that is, it describes the process to be followed for coding any of the modules composing a software system. When needed, this activity can be instantiated in multiple active copies, that are executed in parallel by different SLANG interpreters for supporting the (possibly parallel) implementation of several modules. Figure 6.10 shows a process fragment in which two instances of this activity are invoked. Places **`InfoModuleA`** and **`InfoModuleB`** contain, at the invocation time, the information related with the module to be coded. Place **`GlobalWS`** is shared between the two activity instances, that is, both activities can use and change its contents. **`GlobalWS`** represents a common workspace in which the modules under development are stored in their intermediate and final versions. As for Figure 6.9, the coding activity is started when the information on the module to be coded are available in the input place called **`InfoModule`**. Each module in **`InfoModule`** is characterised by a name, a person responsible for its development, and a list of modules it depends on.

All the modules under development are represented by tokens in the shared place **`GlobalWS`**[9]. Each module in **`GlobalWS`** is characterised by a version number that is increased each time a new version of the interface of the module is developed[10].

The definition of the coding activity supports users in executing the following steps:

a) edit the interface of the module;
b) edit the body of the module;
c) compile the coded module;
d) terminate the execution of the coding activity.

Users issue commands related to these steps using some tools in the SPADE UIE. These commands are received by the coding activity as tokens in place **`UserCommand`**. This is a special place (it can be distinguished by the others for its graphical appearance), called *user place*, as it is used to manage the interaction between the UIE and the Process Engine. A user place can be configured to receive specific types of mes-

9. GlobalWS has associated a type called module. This type characterises all the tokens that can be stored in the place. For the sake of brevity, type definitions and transition codes are just informally described.

10. For the sake of simplicity, in this context, we do not consider the problem of how to manage several versions of the same module. In the example, we just keep track of the number of interface versions that have been developed for each module.

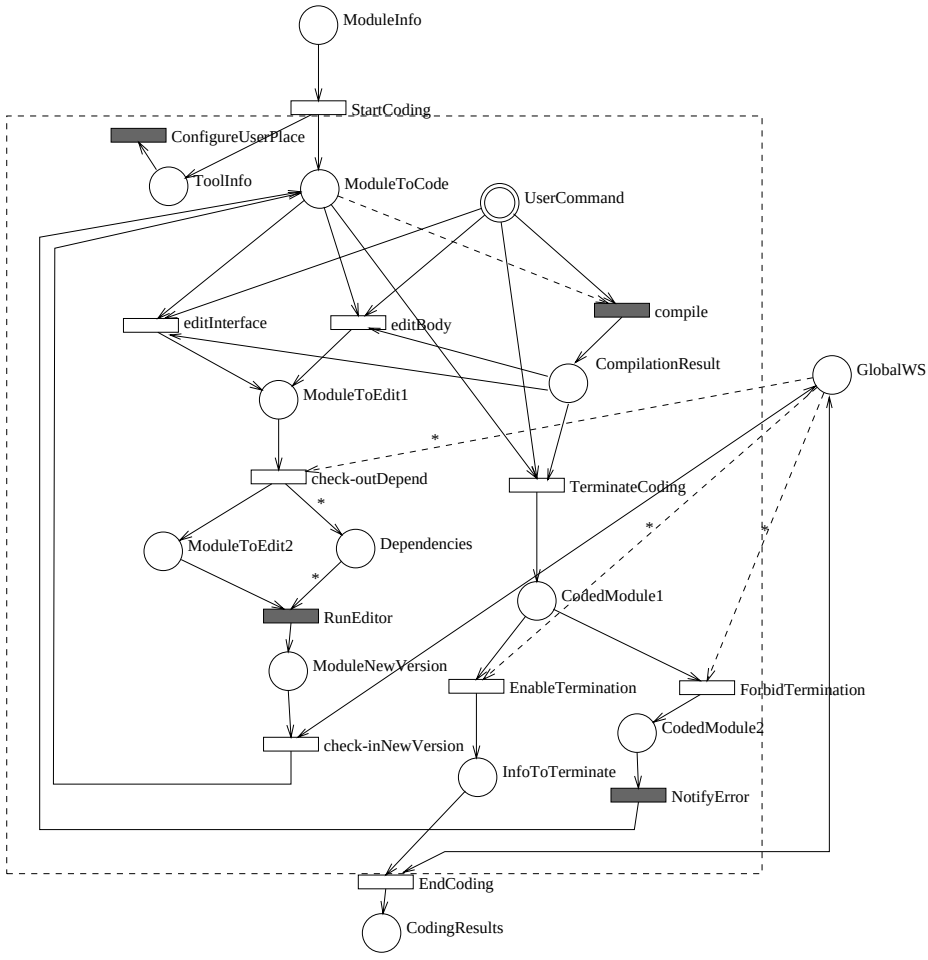

Figure 6.9 The parallel coding activity

sages. In this example, transition **`ConfigureUserPlace`** enables place **`UserCommand`** to receive all the commands of editing, compilation, and termination issued for the module whose development is supported by the activity. The occurrence of a token in **`UserCommand`** enables transitions **`editInterface`**, **`editBody`**, **`compile`**, or **`TerminateCoding`**. They fire provided that their guard is true.

Let us consider the flow of the process when transition **`editInterface`** fires. In this case, a request to edit the interface of the module is represented by a token in place **`UserCommand`** and a token representing the module is in place **`ModuleToCode`** (this happens if no other editing operation on the same module is being performed). The firing of **`editInterface`** starts the editing procedure and enables the execution of **`check-outDepend`**. This last one makes a local copy of all the module interfaces on

which the module to be coded depends. In particular, the last delivered versions of each interface are retrieved from the common repository (**GlobalWS**) and copied in place **Dependencies**. Transition **RunEditor** executes an editor on the user's workstation, thus allowing the user to edit the module interface. The copies of the other interfaces checked out from the common workspace are also loaded in the editor. As soon as the user terminates the editing phase, the new version of the module interface is put in place **GlobalWS** and is made available to all the depending modules.

When the user requests the termination of the coding activity, transition **TerminateCoding** can be enabled to fire. It fires if the module under development has been compiled at least once (a token exists in place **CompilationResult**). After execution of **TerminateCoding**, the guard of transitions **EnableTermination** and **ForbidTermination** are evaluated. The guard of **EnableTermination** is true only if the dependencies (i.e. the module interfaces which the module under development depends upon) have been officially released, and also the module under development has been compiled for the last time using the final release of each dependency. The firing of **EnableTermination** enables the execution of **EndCoding** that marks the last version of the module under development as final, and terminates the execution of the activity. **ForbidTermination** fires whenever **EnableTermination** cannot fire. It enables **NotifyError** that sends an error message to the user. In turn, **NotifyError** inserts the token representing the module under development in place **ModuleToCode** thus enabling the user to issue other editing and compilation requests.

6.4.4.4 Mapping SPADE on the Proposed Framework

In this section we map the SPADE architecture and philosophy on the framework presented in this chapter. In particular, for each of the layers of the framework, the corresponding SPADE elements are highlighted.

Repository: SPADE relies on the O_2 database system for storing both process model descriptions and process artifacts. SPADE users do not interact directly with O_2, but they have visibility on its mechanisms in two cases:

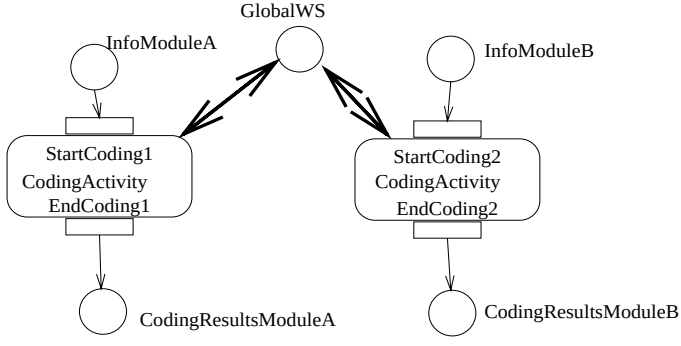

Figure 6.10 Invocation of two active copies of the coding activity

a) In the definition of ProcessTypes and of guards and actions of transitions. In this case they exploit the O_2C language constructs.

b) During the execution of transitions. Each transition, in fact, executes as an ACID transaction exploiting the transactional mechanisms offered by O_2.

Workspaces: In general, a workspace is modelled as a place in a SLANG model. As shown in the example of Section 6.4.4.3 , places can be either private to an activity or shared among activities. In this last case, each sharing activity can read and update the contents of the place. Thus, a common workspace can be naturally implemented with a shared place linked by one or more activity invocations.

Transactions and Knowledge: The granularity of an atomic event in a SLANG model is that provided by the transition construct. The process engine assures that all the transitions that do not involve external events are ACID transactions. SLANG does not offer any primitive mechanism for long or nested transactions. However, such mechanisms can be programmed. From the invoking activity viewpoint, an activity can be viewed as a nested transaction, which in turn may contain both transitions and other activity invocations. In addition, it is easy to add to each activity a net fragment that implements any consistency check before enabling the termination of the activity. In the example (see Figure 6.9), such consistency checks are enabled by the firing of transition **`EndCoding`**. They are needed to verify that the implementation of a module relies on the final versions of all the modules it depends on, and not on some temporary release. This technique can be combined with *shared* places that can be used by one activity to publish the intermediate results of its enactment. Note that SPADE requires an explicit management of consistency and cooperation issues because these "high-level" concepts are not native to the system. Other systems may not require the process modeller to program these parts of the behaviour of the model or they may provide a means to express constraints in a declarative language. The choice of the SPADE system clearly favours flexibility and simplicity, but at the cost of a more detailed model specification. On the contrary, other approaches reduce the process modeling efforts, but at the cost of reduced flexibility.

Interface: The interface layer manages the interaction between the process engine and the human agents who are involved in some way in the process enactment. The interaction with process users is supported by the SPADE communication interface and is always delegated to some tools in the user environment. That is, the user interface *is* a tool that can be controlled by the process model. SPADE provides process modeller with a few integration primitives, a basic set of integrated tools, a set of protocol bridges, and the libraries that are needed to integrate new tools. Like an event driven system, a SLANG model can be programmed to catch a number of events coming from the integrated tools (representing user's actions) and to react to those events according to any particular policy. In the example of Section 6.4.4.3 the coding activity reacts to the users' commands appearing as tokens in place **`UserCommand`**. These tokens enable the execution of transitions depending on the user message they encapsulate and on the internal state of the process.

Interacting with process managers, for control purposes, and with the process modeller, for process debugging, poses different requirements. In particular, it requires the ability to examine the evolution of every step of the model as well as its state. This kind of deep insight into a model that is being enacted is provided in SPADE by means of a privileged system tool called SPADE Monitor.

6.4.5 Other Facets of Cooperation

The advanced transactions models presented in this chapter provide good support for the execution of activities that require for non real-time coordination and data sharing. However, they fail whenever activities like meetings and cooperative editing of documents are to be supported. Such activities pose particular requirements on the architecture of the PSEE. For instance, the PSEE should provide mechanisms with strict synchronisation requirements to multicast data among the participants.

CSCW (Computer Supported Cooperative Working) environments address these issues. However, they usually support specific activities and are not well suited when such activities need to be executed within the context of a more complex and articulated process. SPADE addresses this issue by integrating CSCW tools that extend its cooperation support abilities to synchronous cooperation [Band96].

6.5 Conclusion

This chapter has illustrated the complexity of cooperation control in software processes and the variety of approaches which can be followed to solve the related problems.

Other work has been done on this topic in the context of Promoter: the EPOS approach can be compared with that of ADELE. ProcessWise and SOCCA model cooperation from scratch as SPADE does, but using different formalisms, objects for ProcessWise, state transition diagrams for SOCCA.