Universitetet i Oslo
Institutt for informatikk

Xymphonic Transactions in Workflow Management Systems

Harald Askestad
haraldas@ifi.uio.no

Siv. Ing. Thesis

22nd July 2004

Abstract

Workflow Management Systems (WfMS) have been developed as a means to coordinate organisational processes. In order to provide fault tolerance and concurrency control, concepts from transaction processing systems have been incorporated in WfMSs. Researchers from the database community have attempted to define workflows in terms of advanced transaction models. Another theoretical approach has been to selectively use concepts from transaction theory to better support existing WfMSs. However, most commercial systems use classical flat transactions where possible, but leave a lot to be desired.

This thesis explores the use of xymphonic transactions to improve transactional support in WfMSs. The Xymphonic Transaction Model was presented in (Anfindsen 1997), and introduces conditional isolation and nested databases. These concepts combine to give transactions the ACCID properties, which are better suited to supporting collaborative work.

The thesis proposes and discusses a design for implementing xymphonic transactions in a WfMS. The primary benefits are support for an undo functionality using the transaction manager's recovery mechanisms, and the possibility of grouping multiple and more complex tasks into atomic units. The transactional mechanisms may mostly be automatically enforced and are controlled by the workflow designers through a very simple extension to the workflow definition language. The limitations to the design are considered. Xymphonic transactions are not suitable for environments where many heterogeneous and autonomous systems are to be integrated by the WfMS.

Acknowledgements

During the course of working with this thesis, I have had three supervisors. I would like to thank them all for their support:

Geir Waagbø, who has inspired a pragmatic attitude to information systems, and who has spent much time on helping me, even after he officially quit as supervisor.

Ole Jørgen Anfindsen, who created the Xymphonic Transaction Model and continued as supervisor even after quitting his post at the university.

And Ragnar Normann, who took on the task of internally supervising this thesis when Anfindsen quit, even though he obviously had enough work already.

I would also like to thank Marit Vaksvik, whose cand. scient. thesis has been an invaluable starting point for my own work.

Finally I must express my gratitude for the time and resources that Computas AS and the Norwegian Courts Administration have made available for my case study.

Harald Askestad

Oslo, July 21, 2004

Contents

Chapter 1

Introduction

1.1 Database systems

Database systems have reached an enormous popularity and widespread use since the development of the first general purpose systems in the 1960's. A number of properties contribute to this success. Using a database system, data may be shared, security restrictions may be applied, inconsistency and redundancy can be avoided, and the data may be recovered in the event of a failure (Date 2000).

A database is a collection of persistent data. A database management system (DBMS) is a collection of programs that enables users to create and maintain a database (Elmasri & Navathe 2000). A fundamental property of a DBMS is that it provides data independence. Date defines data independence as "the immunity of applications to change in physical representation and access technique" (Date 2000, p.20). This makes changes to the physical arrangement of data possible, without requiring a rewrite of the applications that use the database. It also allows the database administrator to define different views of the data for different users.

Another abstraction that has contributed greatly to the success of database systems is the concept of a transaction. A transaction is a unit of operations, which must either be executed in its entirety or not at all (i.e. it is executed atomically), it is assumed to be consistent, it must be executed isolated from concurrent transactions, and, once committed, the effects of the transaction must be durable even in the event of serious failures. This gives us four properties — atomicity, consistency, isolation, and durability — that a transaction must satisfy, and they are summed up in the acronym ACID.

Any decent DBMS must provide transaction management. Gray & Reuter (1993) point out that many projects have failed to provide fault tolerant systems without the concept of a transaction. They extend the notion of a transaction processing system to encompass both database systems as well as the applications, developing tools, networking and operating systems software, providing a complete environment for programming, management and use.

Despite the power of transaction processing systems, there are application domains where the semantics of the ACID properties are unsuitable. Isolation contradicts the need for interaction between users. The implementation of transaction management relies on locking the data that is accessed. Applications that require long-lasting transactions will lock the data for an extended period of time, preventing others from accessing those data objects. In most cases it is not possible for more than one application process to operate within the context of a given transaction, effectively preventing users from cooperating on a set of uncommitted data.

Many extensions have been proposed in order to cope with these difficulties and to extend the usefulness of DBMS to new application domains. One of these extensions, the Xymphonic Transaction Model, is the focus of this thesis. It was developed and published in a doctoral thesis by Anfindsen in 1997 under the name of Application Oriented Transaction Model (APOTRAM) (Anfindsen 1997). In this model, interaction between ongoing transactions is allowed by reducing isolation in a controlled fashion. This is combined with the option of creating subtransactions to control the commitment of partially finished work. Together these two mechanisms enable cooperation between users of transactions that run for a long duration.

This thesis explores the applicability of the Xymphonic Transaction Model in workflow management systems.

1.2 Workflow Management Systems

A workflow management system (WfMS) can be characterised as an advanced scheduler that coordinates the execution of activities running on different processing units (Rusinkiewicz & Sheth 1995). These systems may be DBMSs, mailing systems, text-editors, or other stand-alone systems that originally were made to support some aspect of an organisation's business.

The advent of DBMSs in the 1960's and 70's introduced data independence, and separated the concern of handling data from the concern of writing application programs. With WfMSs, business processes are pushed out of the applications. van der Aalst (1998) identifies both of these developments as a trend in going from special purpose information systems towards a more general-purpose software system that can be configured for many different scenarios. The definition given in (Lawrence 1997) illustrates this perspective: "A workflow management system (WfMS) is a generic software tool which allows for the definition, execution, registration and control of workflows."

The introduction of management philosophies such as Business Process Reengineering (BPR) in the 1990s stimulated organisations to become more aware of their business processes. Workflow technology enables the organisation to control and enforce the identified and reengineered business processes (Reijers 2003, p.15). Furthermore, the last decades, organisations often support an increased number of business processes, and at the same time the complexity of the processes has increased and the lifetime shortened. The need for efficient management and rapid changes makes business processes an important issue for current information systems (van der Aalst 1998, p.3).

Transaction management for WfMSs was an active research area in the 1990s. Some fundamental problems were frequently discussed — business processes last for a longer time than what most transaction systems are able to support, and processing is often distributed over heterogeneous and autonomous nodes. Additionally, workflow definition languages are able to express more complicated process patterns than transaction models have been able to support. In fact, (Alonso, Agrawal, Abbadi, Kamath, Günthör & Mohan 1996) indicate that in many aspects, workflow models are a superset of advanced transaction models.

Bearing these difficulties in mind, xymphonic transactions may still be expected to provide some benefits to WfMSs. The Xymphonic Transaction Model supports long-lasting activities. It allows for interaction between ongoing transactions, which might compensate for the general lack of expressiveness. And it might allow new, and previously impossible extensions to the workflow models by supporting more complex tasks in which multiple users collaborate. We may not, however, expect it to solve the problem of heterogeneous and distributed systems. It is the aim of this thesis to discover both the benefits and the drawbacks of using xymphonic transactions in WfMSs.

1.3 Research issues

The following are the research questions that have guided the work.

- *Which benefits can be had from using xymphonic transactions in a workflow management system (WfMS)?*

  This is the primary motivation. The *benefits* will be evaluated by comparing a proposed design with a WfMS that uses classic flat transactions to manage its persistent data.

  In particular, the goal is to support tasks of a longer duration without compromising parallelism, grouping more than one task into an atomic action, undoing tasks by means of the transactional recovery mechanisms and adapting the workflow on the fly to unforeseen situations.

- *How must the WfMS be designed to achieve the promised benefits?*

  There are three aspects of design that are of particular interest. The first is the transaction dimension of architecture. Some WfMSs may incorporate subsystems that do not provide any transactional services. Is it possible to achieve the benefits from using xymphonic transactions in this scenario? And if not, which architecture is the ideal choice?

The next aspect is to consider the structuring of transactions. Xymphonic transactions can be nested in a hierarchical structure. Workflows are designed by routing work in a graph pattern. There may be loops, choices, and branches for parallel execution. Is it possible to find a mapping from workflow design to transaction design?

Finally, parameterised access modes allow xymphonic transactions to interact. How can this feature be utilised? Is there a parameter set that will be generally useful for most workflows?

- *Which factors constrain the use of xymphonic transactions? Which factors limit the applicability of xymphonic transactions?*

  Are there other aspects of a workflow that limit the use of xymphonic transactions? A workflow is a long-running activity. It would be unacceptable to loose several days worth of work due to a transaction abort. Additionally, the tasks may include real actions that cannot be undone. It is often a requirement that the system durably archives the documents and other data pertaining to real actions. And finally, even though xymphonic transactions allow sharing of data between long-lasting transactions, there may be situations that require even more interaction than what xymphonic transactions are able to provide. These are factors that limit the possible duration of transactions.

1.4 Approach

A starting point for working with this thesis was the cand. scient. thesis by Vaksvik (2002). While exploring the desired improvements for workflow systems developed by Computas, she found that advanced transaction models might provide useful functionality. The requirements she identified include undo functionality of completed tasks, making atomic tasks longer than what is possible with the current systems, and grouping multiple tasks into atomic units (*ibid*, pp.66–67).

It has been my aim to extend the work started by Vaksvik. Following her suggestions for further research, I have studied the workflow system LOVISA, being developed for the Norwegian Courts of Justice. This case study has provided information on the transactional requirements of workflows and serves as an example throughout the main discussion of this thesis.

LOVISA also serves as a test of the developed design. The systems that are discussed in this thesis would be far to big to program and test within the limited time available for writing this thesis. However, by predicting the impact my design ideas would have, if implemented in LOVISA, it provides an indication of the soundness of the proposals.

Apart from this input from the case study, the work with this thesis has largely been theoretical. Research in transactional workflows has highlighted some design directions that have proved too limiting for workflows. However, Worah & Sheth (1997) concludes that selected concepts from advanced transaction models will be useful for supporting WfMSs. Consequently, the designs proposed in this thesis are intended to support workflows as far as possible, but whenever the Xymphonic Transaction Model proves to be too limited, they depart from a strict transactional paradigm.

1.5 Document Structure

The chapters 2 and 3 give an introduction to transaction theory and workflow management systems respectively. They provide a basic background to readers that are unfamiliar with these topics.

The case study is described in chapter 4. This provides details of an example workflow management system, which is referred to throughout the discussion.

The discussion itself, in chapter 5, starts with a consideration of architectural issues, and a discussion of constraints to the maximum possible duration for transactions. Next, the main issue of this chapter is a discussion of how the workflow engine may automatically map from workflow definitions to transactions. The discussion is completed by some remaining topics — different uses for conditional isolation, extensions to workflow definition languages, some suggestions for other uses of xymphonies in WfMSs, and a comparison with related research.

Finally, chapter 6 summarises the thesis and concludes with an evaluation of the benefits and drawbacks of the proposed design.

At the end, an index is provided to allow easy reference to the definitions of terms and abbreviations.

Chapter 2

Transaction Models

This chapter gives a theoretical background on transactions. It provides a basis for understanding the discussion in this thesis.

The properties of the classical model of flat transactions are discussed in some detail. This is followed up by briefly introducing some of the developments in advanced transaction models. Finally, the most important part is the description of the Xymphonic Transaction Model. Readers who are familiar with classical transaction theory may skip directly to this section beginning on page 21.

2.1 Flat transactions

The classic transaction is *flat* because it has no internal structure as the user (usually the application programmer) sees it. A transaction is started by using the keyword *begin transaction*. It is terminated by either a *commit* or an *abort*. The set of operations between these keywords is protected by the transaction processing system. The operations are executed sequentially (still this is as seen from the user's perspective), and if the commit is acknowledged by the system, the operations have completed successfully and any updates are guaranteed to be durable. If the application issues an abort, or the system for some reason aborts it, then the system guarantees that none of the operations will have any affect, neither on the database, nor on any other processes in the system. This behaviour is described by the ACID properties.

2.1.1 The ACID properties

Flat transactions are characterised by being atomic, consistent, isolated, and durable. These properties are summed up in the acronym ACID, which was coined by Härder & Reuter (1983) in an attempt to "establish an adequate and precise terminology":

Atomicity: "Either all actions are properly reflected in the database or nothing has happened" (*ibid*, p.289). Atomicity is a property that is relative to the perspective. No operation is truly atomic, but from the perspective of the user, the transaction must behave as if it is.

Consistency: A transaction that commits is assumed to transform the database from a consistent state into another consistent state if it executes all by itself. Constraints defined in the database schema may be checked at the end of the transaction, but it is outside the scope of the transaction processing system to verify that all committed data are truly correct. Thus they are assumed to be, and in the event of a typing error, or a similar fault, it is necessary to correct the mistake with a counter-transaction.

Isolation: "Events within a transaction must be hidden from other transactions running concurrently" (*ibid*, p.290). *Isolation* corresponds to the term *serializability*, which is explained in greater depth below.

Durability: "Once a transaction has been completed and has committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions" (*ibid*, p.290).

The properties are interrelated and easily confused. A transaction that in itself is consistent, will preserve consistency if it executes atomically and in isolation from other concurrent transaction, and if the results are durable. The consistency property, as defined in ACID, is a property of the transaction itself. Consistency of the entire database in the event of concurrency and possible failures is a task, which requires all the ACID properties working together.

The term consistency has also been used to define a correctness criterion for transactions as a whole. Eswaran, Gray, Lorie & Traiger (1976) defined a schedule to be *consistent* if it has an equivalent serial schedule. We will see shortly that the term *serializability* has replaced consistency in this context.

As pointed out in (Anfindsen 1997), atomicity can be confused with isolation. In their discussion of atomicity, Gray & Reuter (1993, p.160) say that "concurrent users must be protected from accessing preliminary data", which implies that atomicity encompasses isolation. This would make the isolation property superfluous. Anfindsen (1997) defines atomicity more precisely to be the atomic *commitment* of a transaction. Once a transaction commits, it "cannot continue to perform work that may or may not be committed at some later point in time" (*ibid*, p.5). Atomicity, as opposed to isolation, is related to recovery, because a transaction that aborts, must recover the data elements that it has modified preceding the decision to abort.

A zipper is a good metaphor for transactions. We may imagine each of the toothed strips to be a transaction, and the teeth themselves to be operations on the database. All the teeth taken together constitute an atomic operation, so we could roll the single strip into an infinitesimally small ball to be executed in an instant. By execution, we could imagine the strip to be sewed to a jacket, and hopefully it would be durable (or else it could be replaced by a backup). We expect the teeth to be in working order, i.e. the transaction is assumed to be consistent. However, we require a transaction to be executed in isolation. Would half a zipper be of any use to us if it were separated from the other half? Let's see how we may "zip" the transactions together, without mixing up the individual operations in an improper order.

2.1.2 Concurrency Control and Serializability Theory

Isolation, as described above, is closely related to concurrency control. A number of transactions that execute one after the other constitute a serial schedule. In a serial schedule, all transactions automatically execute in isolation. A schedule of concurrent transactions is considered to be correct if and only if its effects are the same as the effects of a serial schedule (Garcia-Molina, Ullman & Widom 2002). Stated in another way, a schedule is correct if it is *equivalent* to a serial schedule[1]. In such a schedule all transactions execute in isolation and it is said to be *serializable*.

A schedule, or a history, is an ordering of the individual operations of its constituent transactions. More precisely, Garcia-Molina et al. (2002, p.924) define a schedule of a set of transactions T to be a sequence of actions, in which for each transaction T_i in T, the actions of T_i appear in the same order that they appear in the definition of T_i itself. We are primarily interested in two different operations, read and write. A transaction T_i reading an element x is denoted $r_i(x)$. Conversely, the operation of writing a value to an element is denoted $w_i(x)$. The elements could be tuples, disk blocks, or entire relations. Additionally, the begin of T_i is denoted b_i, and the only possible outcomes are c_i or a_i, which means commit and abort respectively.

Equivalence of schedules is defined in two different ways. Schedules may be *conflict equivalent* or *view equivalent*. Conflict equivalence is the basis for most transaction processing systems, and will be treated first.

A schedule is *conflict serializable (CSR)* if it is *conflict equivalent* to a serial schedule. Two schedules are conflict equivalent if

1. they contain the same transactions, and
2. they order conflicting operations of non-aborted transactions in the same way.

[1]Bernstein, Hadzilacos & Goodman (1987, p.32) gives a more accurate definition. They state that the *committed projection* of a schedule must be equivalent to a serial schedule. This degree of precision is usually left out in introductory texts on serializability, e.g. (Elmasri & Navathe 2000, Garcia-Molina et al. 2002, Anfindsen 1997)

Two operations *conflict* if they are executed by different transactions, they operate on the same element, and at least one of the operations is a write. This gives us four possible combinations, which are summed up in table 2.1. In this table, conflict is marked by two swords. This table is often called a compatibility matrix, referring to the compatibility of the pair of operations that is marked by a smiling face.

	r	w
r	☺	⚔
w	⚔	⚔

Table 2.1: Conflicting operations (compatibility matrix)

The term "conflict" is actually too dramatic. It is perfectly normal to have conflicts. The important point in the definition of conflict serializability is that all the conflicts must occur in the same order as in the equivalent serial schedule. We introduce the precedence graph in order to reason more easily about this order.

A *precedence graph* for a schedule S is a directed graph whose nodes represent the committed transactions $T = \{T_1, ..., T_n\}$ of S. There is an edge from T_i to T_j $(i \neq j)$ whenever one of T_i's operations precedes and conflicts with one of T_j's operations in S. An example schedule and its precedence graph is shown in figure 2.1. The edge from T_1 to T_2 is established by the operation $r_2(y)$, which comes after T_1 has written a value to y. The operations $r_1(x)$ and $r_2(x)$ do not conflict, thus they can be swapped without affecting serializability. Most texts on transactions provide numerous other examples.

$$\begin{tikzcd}
\boldsymbol{T_1} \arrow[r] & \boldsymbol{T_2}
\end{tikzcd}$$

$$S : b_1, b_2, r_1(x), r_2(x), w_1(y), r_2(y), c_1, c_2$$

Figure 2.1: A schedule S and the corresponding precedence graph

The serializability theorem states that a schedule is conflict serializable if and only if its precedence graph is acyclic. This has been proved in (Garcia-Molina et al. 2002, Bernstein et al. 1987) among others. To check for serializability, first derive the precedence graph from the committed transactions of a schedule, and then run a depth first search to check for cycles. While this certainly could be exploited in a real system, more efficient methods exist. The most common method is two phase locking, which is explained in section 2.1.5.

Extending the zipper metaphor, we would say that the two strips taken together constitute a schedule. It is a schedule that adheres to the definition above — no teeth can be added, or removed, without disrupting the zipper, and they will appear in the schedule in the same order as they appear in each individual strip. If we hold one strip entirely before the other, the zipper is a serial schedule. To restrict us to serial schedules is just as limiting for real transactions, as it is for the zipper. A transaction processing system enforcing serial schedules only would be unacceptably inefficient.

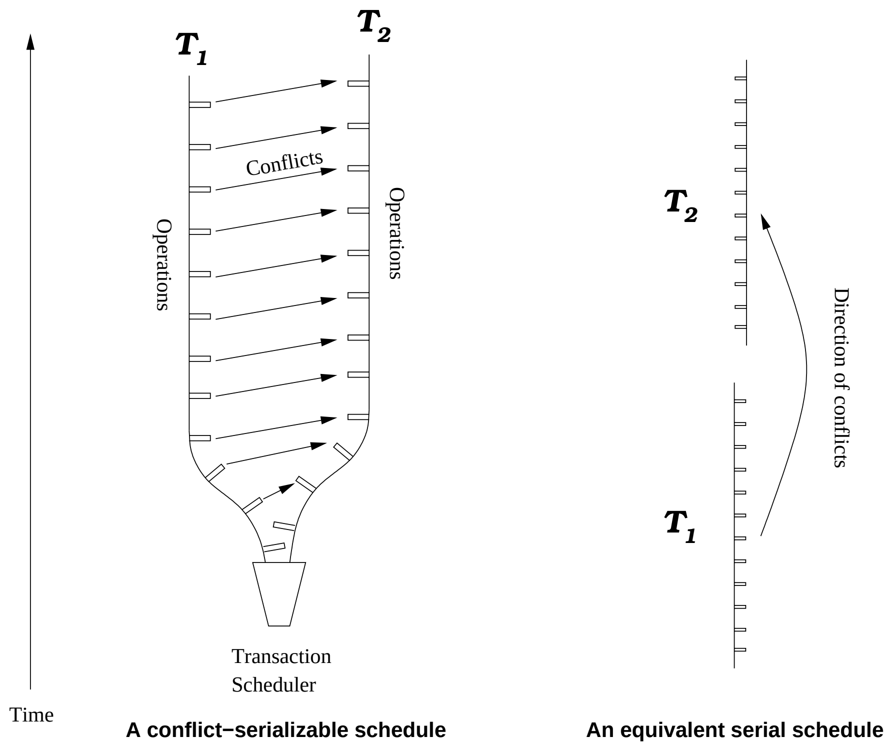


Figure 2.2: Conflict serializability visualised as a zipper

Now, let's zip up the transactions, and imagine that each pair of teeth from the two strips is a pair of conflicting operations in the way shown in figure 2.2. The arrows represent the conflicts in the same direction as it would be represented in a precedence graph, i.e. transaction T_2 does an operation, which is in conflict with a previous operation on the same object by transaction T_1. Then the schedule on the left would be conflict equivalent to the serial schedule on the right. The serial schedule is created by unzipping, and moving the two parts in the temporal direction of the arrows.

The zipping is controlled by the sliding tab, which is marked as transaction scheduler. If we try to bypass a tooth, the zipper will easily fall apart. This is illustrated in figure 2.3, which shows a tooth out of order, thereby creating conflicts in the other direction. That is, T_1 performs an operation which is in conflict with a previous operation by T_2. Such behaviour creates a loop in the precedence graph as shown, and must be prevented by the scheduler.

It is important to note that the zipper metaphor represents two transactions that have the same number of operations. Real transactions will rarely have such symmetry,

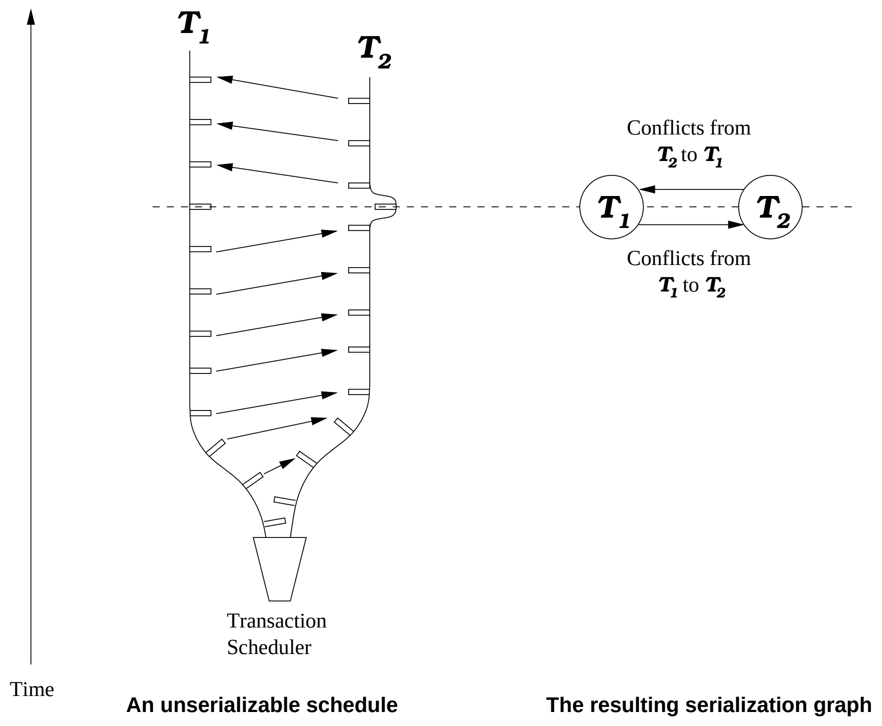

T_1
T_2
Conflicts from T_2 to T_1
Conflicts from T_1 to T_2
Transaction Scheduler
Time
An unserializable schedule
The resulting serialization graph

Figure 2.3: Unserializable behaviour

and not all operations will conflict. The scheduler will normally interleave more than two transactions, and each transaction may be allowed to execute more than one operation before the other transaction is allowed to run. Thus the metaphor represents a special case that is useful for illustrating the principles, but it must not be taken as a typical example.

2.1.3 View serializability

View equivalence is another, less restrictive way to define equivalence of schedules. To define this, we say that a transaction T_i *reads from* a transaction T_j if T_i reads a value that T_j has written, and T_j has not (yet) aborted.

A schedule is *view serializable* if it is *view equivalent* to a serial schedule. Two schedules, S and S', are view equivalent if

1. they contain the same transactions, and
2. if T_i reads x from T_j in S, then it must also be the case that T_i reads x from T_j in S', and
3. if the last transaction to write x in S is T_i, then T_i must also be the last transaction to write x in S'.

(Anfindsen 1997, p.8)

Recall that in the general definition of serializability, a schedule is considered to be correct if its effects are the same as that of a serial schedule. The computation that a transaction does is based on the values that it reads from the database. Point 2 above simply states that this computation will be the same if the values read by the transaction are the same in both schedules. Thus the calculation of the value to write to the element x will also be the same, and as long as the last transaction to write this value is the same in both schedules, the effects will be the same.

The set of conflict serializable schedules is a proper subset of view serializable schedules. However, algorithms for enforcing view serializability are NP-complete, so they are not used in commercial systems.

2.1.4 Recovery related properties

Atomicity and durability are closely connected to recovery. If a transaction aborts, it is necessary to roll back, or undo whatever effects it has caused so far. And in order to enforce durability, if the system fails, committed transactions may have to be recovered. Until now we have not considered the abort and commit operations, but they are of critical importance. If the commits and aborts are executed in the wrong order, the system may not be able to recover the database to a consistent state, even though the schedule may be serializable.

The problem can be illustrated by considering the serializable schedule in figure 2.2. If transaction T_2 commits before T_1 terminates, then aborting T_1 could lead to inconsistencies. If T_2 has read from T_1 and therefore based its computing upon these values, it should also have been aborted. Such a schedule is not recoverable.

There are four increasingly restrictive classes of recoverability that may be imposed on a schedule. The first three are treated in (Bernstein et al. 1987, pp.34–35), and rigorousness was introduced by Breitbart and is treated in (Breitbart, Georgakopoulos, Rusinkiewicz & Silberschatz 1991).

A schedule is *recoverable* (RC) if whenever a transaction T_2 reads from another transaction T_1, then T_2 will only commit after T_1 has committed. If T_1 aborts, then T_2 must also abort.

RC is a minimal requirement for providing atomic transactions. It prevents a transaction that has performed a dirty read from being committed until the transaction it read from is committed as well.

A schedule *avoids cascading abort* (it is ACA) if whenever a transaction T_2 reads from another transaction T_1, then the read operation of T_2 must wait until after T_1 has committed.

ACA prevents dirty reads. As the name implies, ACA will prevent establishing dependencies between transactions that cause an abort of one transaction to propagate to the entire set of dependent transactions.

A schedule is *strict* (ST) if whenever a transaction T_2 performs an operation on an element written by another transaction T_1, then the operation must wait until after T_1 has committed.

ST simplifies recovery by allowing the use of before images (BFIM). A BFIM is the original value of an element x that is updated, and it is usually recorded in the log. To undo the update, the BFIM may be written to x. ST guarantees that no other transaction may update x, and thereby invalidate the BFIM, until the original transaction is either committed or aborted.

A schedule is *rigorous* (RG) if whenever a transaction T_2 performs a conflicting operation on an element read or written by another transaction T_1, then T_2's operation must wait until after T_1 has committed.

RG will guarantee that the serialization order is analogous to the execution order of the transactions (Breitbart et al. 1991, p.956). This is an important mechanism for enforcing global serialization in distributed transactions.

The definitions are increasingly restrictive such that RC$\supset$ACA$\supset$ST$\supset$RG. If there is a write-read conflict between active transactions, RC defines the order in which transactions may commit. ACA on the other hand, will prevent write-read conflicts entirely, and therefore it is also RC. ST, in addition to preventing write-read, also prevents write-write conflicts. RG additionally prevents read-write conflicts. The definitions are summed up and compared in table 2.2. In this table an operation, which could be either a read or a write, is denoted o.

RC	if $w_1(x)$ precedes $r_2(x)$, then c_1 must precede c_2
ACA	if $w_1(x)$ precedes $r_2(x)$, then c_1 must precede $r_2(x)$
ST	if $w_1(x)$ precedes $o_2(x)$, then c_1 must precede $o_2(x)$
RG	if $o_1(x)$ precedes and conflicts with $o_2(x)$, then c_1 must precede $o_2(x)$

Table 2.2: Increasingly restrictive properties related to recovery

Returning to the zipper metaphor, we may classify the conflict serializable schedule in figure 2.2 in terms of the new properties. Assuming the termination of T_1 is a commit, the schedule is RC. The schedule is ACA if, for all the writes performed by T_1, then the conflicting operation of T_2 is also a write. If all the operations of T_1 are reads (and all the operations of T_2 must be writes, otherwise they would not conflict), the schedule is ST. However, the schedule cannot be RG because the conflicting operations of T_2 come before T_1 has committed. A rigorous scheduler would only allow the serial schedule to the right in this figure.

2.1.5 Enforcing Flat Transactions

Transaction processing systems commonly control concurrency by locking. The general idea is that a transaction T_i must acquire a lock on an element before it can process it. If another transaction holds an incompatible lock, then T_i must wait. The transaction T_i must hold the lock at least until it is finished processing the element. The schedule will be conflict serializable if the transactions follow the two phase locking (2PL) protocol: Once a transaction has released a lock, it cannot acquire any more locks (Bernstein et al. 1987, p.50).

There are two fundamental types of locks, read locks and write locks. To read an element x, a transaction must at least set a read lock on x. To write, a transaction must set a write lock on x. Two locks are compatible if their corresponding operations do not conflict. Recall table 2.1 that illustrated conflicting operations. This table illustrate compatibility of locks by a smiling face.

The correctness of 2PL is easily explained by considering the precedence graph of a schedule. An edge from transaction T_i can only be established if it has released a lock, and some other transaction sets a conflicting lock. This holds inductively, so a transaction T_j can only follow T_i in the graph, if it has acquired an incompatible lock that its predecessor in the path from T_i has released. To establish a cycle, T_i would have to set a lock that conflicts with a lock that T_j has released. This is not possible because 2PL prohibits T_i from acquiring any more locks.

Basic 2PL can easily be extended to support recovery. Strict 2PL requires the transaction to hold all write locks until it terminates, effectively preventing any other operation on the locked elements until it has committed. Note that strict 2PL produces schedules that are both strict and serializable, thus it is more restrictive than strictness taken alone. Rigorous 2PL requires the transaction to hold all locks until termination, preventing any conflicting operations on the locked elements until it has committed. Rigorous 2PL is commonly used, as it is hard to decide when it is possible to start releasing locks.

Other scheduling techniques exist. In timestamp ordering, the transaction manager assigns a unique timestamp to each transaction, and the scheduler order conflicting operations according to the timestamps. The effect is that edges in the precedence graph are only allowed from the older to the younger transactions.

A system based on multiversion concurrency control keeps the old versions of an element that is updated. The scheduler decides which version to use in order to maintain serializability. A read request that would have been denied by a 2PL scheduler, may be possible when the transaction is given the older version. There are multiversion concurrency control algorithms both for ensuring view and conflict serializability.

2.2 Advanced Transaction Models

Flat transactions were designed for quickly updating small amounts of data, and have been highly successful in bookkeeping applications. Other application domains, however, have access patterns that flat transactions cannot support very well. Typical examples of such domains include technical design (CAD/CAM), computer aided software engineering (CASE), office automation, data mining and multimedia.

A transaction model is a specification of allowable and mandatory behaviour for transactions as well as their structure (Anfindsen 1997, p.23). The classic transaction model specifies both simple behaviour and structure.

The selection of transaction models presented here is far from comprehensive. Rather, it is intended as a background for understanding the Xymphonic Transaction Model. This may give the false impression that different extended transaction models are quite similar. As pointed out in (Gray & Reuter 1993, p.180) — many extensions to flat transactions have been proposed to support quite specific applications, however, their usefulness may be limited for other applications, and different types of transactions may not be able to coexist in the same system. No transaction model is yet general enough to accommodate all the different applications' requirements. However, xymphonic transactions incorporate many of the features from the other extensions.

2.2.1 Supporting Long-Lasting Transactions

Many of the advanced transaction models try to support transactions that last for a long time. Such transactions are collectively known as long-lasting transactions (LLTs). Transactions may have a long duration if they access many objects, perform complex operations, or wait for interaction with humans or other external systems.

LLTs introduce particular problems for transaction management:

- Locking resources for a long time prevents others from accessing them.
- Locking many data items increases the risk of deadlocks.
- In case of an abort, the amount of work lost may be great.
- LLTs have higher probability of being interrupted by a system failure or a shut down for maintenance.

Finally there is the question of how to maintain application context. Only the global context is made durable by the DBMS. Applications (and users) have private context that may be lost in case of a crash (Gray & Reuter 1993, pp.212–215). It is necessary to recover this context to a state that is consistent with the state recorded in the database, otherwise the application may proceed in the wrong way.

Gray & Reuter (1993, p.217) discuss a set of requirements for supporting LLTs:

- *Minimise lost work* in case of a program or system crash.
- Provide *recoverable computation* to allow for the system to be shut down and restarted without requiring a commit or abort.
- Provide *explicit control flow* so that the system may control the sequence of transactions belonging to one LLT. “At any point in time, and under all failure conditions, it must be possible either to proceed along the prespecified path or remove from the system what has been done so far.” (*ibid*, p.217)

And, I would add a final point:

- *Allow interaction between ongoing transactions.*

Even a long-lasting activity is regarded as an atomic unit of work (otherwise it could be split into a set of smaller transactions, and the problem would have been avoided). It is required that the system controls the commitment of the transactions and that the changes are made durable at this time. And we would certainly not like introducing inconsistencies. But as the point above shows, reducing isolation may be a desired property. Many extended transaction models reduce isolation in a controlled manner, and Anfindsen (1997, p.7) maintain that isolation is the only property that should be compromised.

In general, reducing isolation in a *controlled manner* means allowing access to uncommitted data in such a way that the desired level of consistency is preserved. The isolation level will be application dependent, thus a mechanism for applications to explicitly control the transaction is needed. We will see one example of how this may be achieved in the discussion of the Xymphonic Transaction Model (section 2.3).

2.2.2 Spheres of Control

Bjork and Davies developed the concept of spheres of control in the 1970s. By watching how human organisations deal with errors and their recovery, they proposed a model for monitoring dependencies between processes and recovery from errors that potentially may have a source in the past. It was “the first attempt to investigate the interrelated issues of concurrency control and recovery in integrated systems in a coherent fashion” (Gray & Reuter 1993, p.160). Spheres of control has never been fully realised, but it lead to the development of the classic transaction model.

According to Gray & Reuter (1993, p.174) spheres of control have two core functions:

1. To contain the effects of operations as long as there may be a necessity to revoke them, and
2. To monitor the dependencies of operations on each other in order to be able to trace the execution history in case faulty data are found at some point.

Davies (1978) describes several types of SOCs. He understands a system as a hierarchy of processes or abstract data types. A process at some level may use operations at the next lower level for implementation, e.g. in the form of a function call. As seen from the level above, the process executes atomically. Atomicity control is a SOC that protects the effects of a process from being externalised before the execution finishes. Davies uses atomicity in the broad sense that encompasses isolation.

Figure 2.4 illustrates some processes and their SOCs. Here we see a SOC containing each of the processes A and B. Nested within them are SOCs that control the atomicity of the lower-level operations that they use for their implementation. The operations A1 and A2 are processed in parallel, and together they form the output from A. The operations nested in B are processed sequentially.

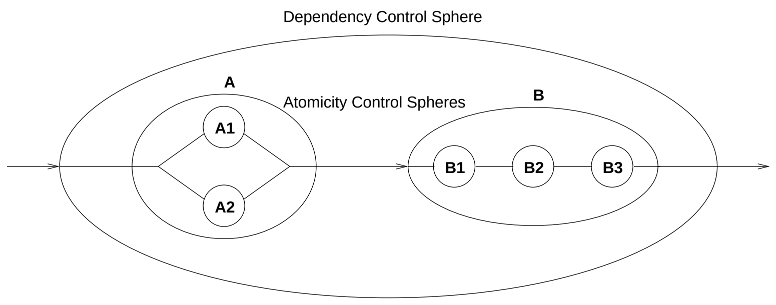


Figure 2.4: Processes A, B and some spheres of control

As other SOCs use data from existing ones, dependencies are established between them. SOCs may be dynamically created around processes to control this dependency. Figure 2.4 illustrates a dependency from processes A to B. When for instance process B reads from A, before A has committed, a SOC is dynamically created to control the commitment, or alternatively, the rollback of both processes as a single unit. This may allow A and B to start executing in parallel, although logically, the execution is serial.

Several SOCs deal with recovery. In-process recovery allows processes to return to a previous point of execution, i.e. to whatever state was recorded as the SOC was created. System recovery is useful for recovering from system failures. And finally, post-process recovery allows a previously committed process to be rolled back. To achieve this, the entire execution history with information of which versions are dependent on each other must be kept. When an error is detected, the recovery operation must trace back to the source of the error and establish a SOC around the originally erroneous process. It then traces forward to extend the SOC to cover the closure of all the other SOCs that have a dependency relation with the source of the error.

A transaction is a very simple form of SOC. The atomicity and isolation properties have their close parallel to the concepts of atomicity, commitment and dependency control. The durability property, however, contradicts the post-process recovery mechanism. It seems practically impossible to maintain enough data to re-establish consistency, and the actual recovery procedure would be dependent on the situation. As an example, Gray & Reuter (1993, p.179) mentions that revoking a fraudulent money transfer, could involve recovering the cash from the responsible person.

2.2.3 Savepoints and Persistent Savepoints

Savepoints enable flat transactions to undo parts of its work and continue execution from an earlier point. The point to which the transaction may return is a savepoint and it is set by the transaction itself. It is useful in situations where the transaction may choose between different paths of execution. If one path does not work, the application could roll back to a savepoint and try another direction from there. This saves the work of starting all the way from the beginning. Savepoints have become an optional feature of the SQL-99 standard (ISO 2000, pp.82–83,721 and 726).

Persistent savepoints address the particular problem of minimising lost work if a system fails during execution of a long-lasting transaction. A persistent savepoint is a savepoint in the sense described above, but the transaction state must be recorded in durable storage. Upon recovery, changes that happened after the savepoint must be undone in the same way as is done for uncommitted transactions, and changes before the savepoint may have to be redone.

A problem with persistent savepoints is that the state of the application program may be lost in case of a failure (Gray & Reuter 1993, p.191). In order to handle this correctly, the application state necessary for continuing execution after the savepoint must be recorded in durable storage as well.

2.2.4 Nested transactions

Savepoints allow for the structuring of a transaction into a sequence of smaller units. Nested transactions allow for the decomposition of transactions into a hierarchy of smaller units. This hierarchy provides fine-grained control over commitment and recovery of different parts of a transaction, it allows distributing the execution of individual subtasks among multiple nodes, and supports intra-transaction parallelism.

There is always a root, or top-level transaction. Within this, subtransactions may be established. The nesting may continue to an arbitrary depth, forming a tree structure of transactions. The usual vocabulary for trees apply to the nodes of the tree, e.g. all transactions apart from the root have a parent, and their subtransactions are referred to as children.

A subtransaction may commit if it is a leaf transaction, or if all its subtransactions have terminated. Upon commitment, the results of the transaction are made available to its parent (and possibly its siblings). Its commitment will not be durable until all its ancestors all the way to the root have committed too. When the top-level transaction commits, it is said to be finally committed. On the other hand, if a transaction aborts, this triggers the abort of all its subtransactions, even those that already have issued a commit.

As seen from the outside, a nested transaction is indistinguishable from a simple flat transaction. The sphere of control established by the top-level transaction covers the whole hierarchy and provides it with the ACID properties as a unit. The individual subtransactions are atomic from the perspective of the parent and their execution is isolated from each other. They are consistent with respect to the local function they implement, although global consistency may depend on the combined effects of all subtransactions. But as discussed above, their durability depend on their parents recursively up to the root.

Nested transactions are generally attributed to a PhD thesis by J Eliot B Moss in 1981. As can be seen, the ideas of nesting are quite similar to those of spheres of control, but Moss' system was the first to use locking for synchronisation (Moss 1985, cited in Anfindsen 1997).

In Moss' original scheme, the commitment of a subtransaction would pass its locks to the parent, so called *upwards inheritance*. Härder & Rothermel (1993) extend this with the concept of *downwards inheritance*. They distinguish between *holding* and *retaining* locks. A transaction that acquires a lock on an element x in the traditional sense, holds a lock. If a transaction retains a lock, its children will be able to acquire it, but any transaction outside the subtree (including ancestors) will be prevented from acquiring a conflicting lock. The locks of a committing subtransaction will automatically be retained by its immediate parent (i.e. upwards inheritance).

Multi-level transactions are a generalised form of nested transactions (Gray & Reuter 1993, p.203). They differ from the latter in that subtransactions are allowed to commit, but their results are protected from externalisation by enforcing a strictly layered architecture. As long as the parent locks higher-level data, no other transaction may access the precommitted lower level data structures. In the event of an abort, the precommitted subtransactions may be undone by executing a compensating transaction without affecting serializability.

Gray & Reuter (1993, pp.206–210) discuss the applicability of multi-level transactions in the implementation of DBMSs. They are useful when the granularity of locking at the physical level is required to be greater than what is needed by an operation, e.g. when updating only a single tuple on a disk block. Precommitting the subtransaction allows other tuples on the same block to be accessed, but the parent's locking of higher-level access structures precludes other transactions from accessing the updated tuple.

2.3 The Xymphonic Transaction Model

The Xymphonic Transaction Model was developed and published in a doctoral thesis by Ole Jørgen Anfindsen in 1997 under the name of Application Oriented Transaction Model (APOTRAM) (Anfindsen 1997). The author has implemented the model in a commercial system called the Xymphonic Engine. Different aspects of the model have been studied in several Cand. Scient. and Siv. Ing. Theses (Vaksvik 2002, Kjølstad 2001, Sommerfelt 2001). APOTRAM as it is presented in (Anfindsen 1997) is mainly a model. Although the PhD thesis considers implementation issues, the actual functionality of the Xymphonic Engine may not cover all the aspects of the model, and it may also provide new options that were not apparent as the model was developed at first. This section integrates the new developments with the presentation of the model as it was originally conceived.

2.3.1 Allowing interaction between ongoing transactions

Isolation and interaction may be regarded as opposites on a scale showing the degree to which active transactions interact[2]. ACID transactions operate in total isolation — they have no interaction with concurrent transactions. The only interaction is with the database, which is actually the committed transactions of the past. Reduced isolation allows a greater degree of interaction. The simplest scheme is called uncommitted read (UR) and allows a transaction to read from other uncommitted transactions. However, the schedules produced will in general be non-serializable and non-recoverable. UR and other isolation levels are defined in the SQL-92 standard, but although the stricter schemes may eliminate some of the problems associated with UR, the active transaction has no way of knowing how reliable the data is.

The approach taken in xymphonic transactions is to give the user, represented in the system as a transaction, several mechanisms for fine-tuning the degree of interaction. In order to let others read uncommitted data, the Xymphonic Transaction

[2]The idea of using the term *interaction* comes from talks with O. J. Anfindsen.

Model introduces the notion of conditional conflict. Using a parameterised access mode, a transaction may associate a set of write parameters, taken from a predefined domain, to each write operation. The data item is locked according to the normal compatibility matrix for two phase locking. However, another user or application program may associate a set of parameters with the read operation, thereby signalling to the scheduler that it is acceptable to read uncommitted values whose write parameters match those of the read operation. The decision whether the write and read operation conflict is made conditionally dependent on the access parameters. This results in the correctness criterion called *conditional conflict serializability*, which is defined in section 2.3.2.

Collaborative writing of the same set of uncommitted data is achieved by combining parameterised access modes with nested databases. A user, or application program holding the write locks of a document, may convert the set of locks to a nested database or a xymphony. Other users may be invited to create subtransactions in the database, update the data, and when finished, commit to the nested database. Finally, when finished, the original user may commit the nested database as he would an ordinary transaction. Nested databases are treated in section 2.3.3.

In the model outlined, intertransaction interaction is increased by *a)*, allowing others to read uncommitted data under certain conditions, and *b)* allowing other users to work on and complete part of the work.

2.3.2 Conditional Conflict Serializability

The Xymphonic Transaction Model introduces the concept of *parameterised access modes* and *conditional conflict*.

Parameterised access modes means that an access request is associated with a set of parameters. The parameters are taken from a domain of values that are defined by the application developers (or possibly by the users themselves, depending on the type of application). For example the parameter domain {draft, preliminary, proposed, accepted} may indicate the status and reliability of a document in the database. Parameter domains may be defined to accommodate other, application specific requirements, e.g. to indicate quality, maturity, reliability or degree of completeness (Anfindsen 2002, p.4).

With conditional conflict, conflict is conditionally dependent on the parameters of read and write operations. Let $R(A)$ and $W(B)$ denote a parameterised read and write operation respectively, where A and B are the associated parameters. Both A and B must be taken from the domain D of available parameters. Then conditional conflict is defined as follows.

Definition 1 (Conditional Conflict) *The parameterised read mode $R(A)$ and the parameterised write mode $W(B)$ conflict unless $B \subseteq A$.*

(Anfindsen 1997, p.29)

Or stated the other way around, $R(A)$ and $W(B)$ are compatible iff B⊆A. This gives the revised compatibility matrix in figure 2.3.

	$R(A)$	$W(B)$
$R(A)$	☺	☺ iff $B \subseteq A$
$W(B)$	☺ iff $B \subseteq A$	⚔

Table 2.3: Conditionally compatible operations

Two schedules are defined to be *conditionally conflict equivalent* the same way as they are conflict equivalent, provided table 2.3 is used to determine which operations conflict (see page 9 for the definition of conflict equivalence). The new correctness criterion is based on this notion of equivalence.

Definition 2 (Conditional Conflict Serializability) *A schedule is defined as conditional conflict serializable (CCSR) iff it is conditional conflict equivalent to a serial history.*

(Anfindsen 1997, p.30)

CCSR transactions reduce isolation by making it *conditional*. Thus the acronym ACCID describes their properties. That is, xymphonic transactions have the properties atomicity, consistency, *conditionally isolation* and durability.

ACCID transactions may coexist in the same system as flat CSR transactions. The default is to treat a normal read operation as equivalent to a parameterised $R(\emptyset)$, and a write as a $W(D^*)$ operation, where D^* is an arbitrary superset of the parameter domain D. This means that a non-parameterised transaction will keep its ACID properties, while at the same time there may be some transactions that reduce the isolation among each other. Thus CSR is a special case of CCSR.

By using parameterised write modes, a user declares a willingness to share that data before the computing is finished. Another user issues a parameterised read mode to declare a willingness to read data that may not be committed. If the parameters of the read request is a superset of the parameters of the write request, the system will grant the read request. Thus the meaning of the parameters is defined by the users, not the system. Note that because the compatibility matrix is defined to be symmetric, a parameterised read mode also means that a data item that is read by one transaction T_1, may be overwritten by another transaction T_2 before T_1 has started releasing its locks. We may say that a parameterised read request also signals a willingness to let others modify the data that has been read.

2.3.3 Nested Databases

CCSR deals with read-write conflicts. The concept of *nested databases* was introduced in (Anfindsen 1997) in order to manage write-write conflicts. Nested databases let multiple users collaboratively write data that is protected by a single transaction. Combined with CCSR, other users may review the progress of the work.

The locks of a transaction can be seen as a sphere of control (SOC). A set of write locks is denoted WSOC. A WSOC is similar to a database within which a single transaction is active. The WSOC prevents outsiders from accessing the protected objects, and because only one transaction is active, no concurrency control is needed within. Nested databases allow a WSOC to be converted into a subdatabase, thereby establishing access and concurrency control similar to that of the parent database. Multiple transactions may be started in parallel within the scope of this database, and these in turn, may be converted to new nested levels of database.

A subdatabase is treated like a nested transaction, albeit a passive one. In the manner of ordinary nested transactions (see section 2.2.4), a transaction in a nested database will commit its results to the parent database, and the results will be finally committed if the top-level database commits its results to the system. When committing, the subtransaction prepares to commit, and the WSOC of the transaction is handed (back) to the parent (upwards inheritance). The subdatabase owner is now free to pass the locks on to some other subtransaction that may wish to continue the work (downwards inheritance).

A nested database is controlled by the transaction owner. In the manner of ordinary databases, an owner may specify access privileges. He may specify which objects may be read, or updated and by whom. We say that the database owner *invites* other users to participate and to complete a part of the whole project. Due to recursive nesting, other users may in turn convert their part of the job to a database and allocate responsibilities to other participants. When a subtransaction prepares to commit, the subdatabase owner may opt to review the changes before they are accepted.

Enforcing conflict serializability in all nested databases gives a schedule that is *nested conflict serializable* (Anfindsen 1997, p.49). When combining nested databases with CCSR, we get the correctness criterion *nested conditional conflict serializability*, which is defined as follows.

Definition 3 (Nested Conditional Conflict Serializability) *If subdatabases can be nested to arbitrary depths, and transaction histories in subdatabases are CCSR, and transactions in subdatabase histories commit to the subdatabase owner, then the resulting transaction schedule will be nested conditional conflict serializable (NCCSR).*

(Anfindsen 1997, p.50)

2.3.4 Querying Unreliable Data and Missing Data

Using parameterised access modes introduces the problem of evaluating queries that may retrieve uncommitted, and therefore unreliable data. To handle this, Anfindsen (1997) proposes Annotated Logic (AL) as an extension to the logic of the query language. Using the same general approach, Anfindsen additionally proposes Nullable Logic (NL), which address the problem of querying missing data. Defining a logic that consistently handles missing data is extensively treated in the database literature (see Anfindsen 1997, pp.62–63 for a quick overview). Combining AL and NL yields Annotated Nullable Logic, a comprehensive approach to querying both missing and unreliable data.

Annotated Logic is sufficient for querying the status of a xymphonic transaction. AL enables a user to construct predicates that evaluate the parameters currently associated with a locked data item. If for example the parameter domain is defined to let transactions indicate the status of the locked resources, AL queries could be used to monitor the progress of the work done in a certain subdatabase.

Nullable Logic allows a set of null marks to be defined and used to indicate why a data value is missing. An example domain of null values could be {does-not-exist, unknown, no-information, prohibited, pending, secret} (Anfindsen 1997, p.64). ANL requires the domains of reliability marks and null marks to be disjoint.

Basically ANL operates with the three truth-values *true*, *maybe* and *false*. The truth-value is associated with a set of annotation marks that are derived from truth-valued expressions, e.g. a comparison like $a > b$. The set of annotation marks associated with a truth-value is defined to be the union of all the annotation marks of the data evaluated in the expression.

The resulting annotated truth-value is denoted (u, A), where u is the truth-value resulting from the comparison, and A is the set of annotation marks. A could be the empty set, meaning that u is either true, or false, and that this is the result of comparing reliable data. If A is a subset of the domain of reliability marks, u is still either true or false, but the evaluation was based on unreliable data. However, if A contains a null mark, the value of u is a maybe (or Boolean null), and the null marks in A will indicate why some of the evaluated data was missing.

(Anfindsen 1997) includes a definition of the effects of the Boolean operators NOT, AND, and OR. The details are omitted, but note that ANL supports the laws of idempotency, commutativity, associativity, universal bounds, De Morgan's laws, and partially the law of distributivity (*ibid*, pp.68–72). ANL does not support the law of absorption, however, Anfindsen comment that it is impossible for a multi-valued algebra to support all the laws of Boolean algebra, and he claims that ANL is better than what is offered by SQL.

2.3.5 The Xymphonic Engine

In conjunction with the commercial implementation, Anfindsen & Storløpa (2001) introduce the term *xymphony* to denote a nested database. Whereas the terms nested database and subdatabase describe the technical properties of the construct, they maintain that the term xymphony communicates better with both users and computer professionals. They define a xymphony as “a database that is, or potentially can be, part of a recursive and dynamic structure of databases”(*ibid*, p.2). The terms xymphony and subdatabase are used synonymously in this thesis.

The Xymphonic Engine was first implemented on top of the Oracle database. A component known as the Internet File System (IFS) intercepts client application calls to the database server and parses the requests. The parser is a Java object that has been extended in subclasses to forward access requests to a separate lock server using the Java Remote Method Invocation protocol. The lock server implements the Xymphonic Transaction Model, allowing parameterised access modes and the creation of xymphonies. The lock server decides whether to grant or deny a request, and the parser implements the decision (Anfindsen & Storløpa 2001, p.3). The implementation has been ported to run on top of other DBMS products as well.

In addition to supporting the Xymphonic Transaction Model, the Xymphonic Engine allows a xymphony to be checked out to another physical machine. The other machine controls access to the resources of the xymphony until the results are checked back in as the result of a commit or abort.

Collaboration between different xymphonies is further supported by the possibility of exporting a selected set of data, which may be imported read only in another xymphony. This corresponds to using write and read parameters that let other transactions read, but not modify uncommitted data. However, the activation of an export function allows better control of when and to whom the data is visible, and the user may continue working on the exported data, without affecting the version that other users import. Additionally, an export function is more easily understood and the actual manipulation of access parameters may be hidden by the system.

Chapter 3

Workflow Management Systems

Computerised workflow management is based on an explicit workflow model that is represented as data in the system and interpreted by a workflow engine. The model includes several process definitions, each consisting of tasks to be executed for different types of cases, and the order in which to do them. The process model refers to an organisation model in order to select participants and other resources for a task. In this way, a WfMS support the flow of work through the organisation.

The Workflow Management Coalition (WfMC) was founded in 1993 and is an organisation of workflow vendors, users, analysts and university/research groups. Their expressed mission is to establish "standards for software terminology, interoperability and connectivity between workflow products"[1]. The standards they propose are an important basis for the discussions in this thesis. The workflow reference model and the meta model for process modelling will be presented in the following two sections.

3.1 The Workflow Reference Model

The workflow reference model (WfMC 1995) and its associated terminology and glossary (WfMC 1999*b*) provide the context for the standards from the Workflow Management Coalition. They identify the characteristics, terminology and main functional components of a WfMS. The individual components are further detailed in a series of documents.

The workflow reference model gives an overview of the major components and interfaces within a workflow architecture. It is shown in figure 3.1 on the next page. Products may vary in how many of the interfaces they expose. A fully conforming product would be able to substitute any of the subsystems with a corresponding component from another vendor.

[1]http://www.wfmc.org/about.htm

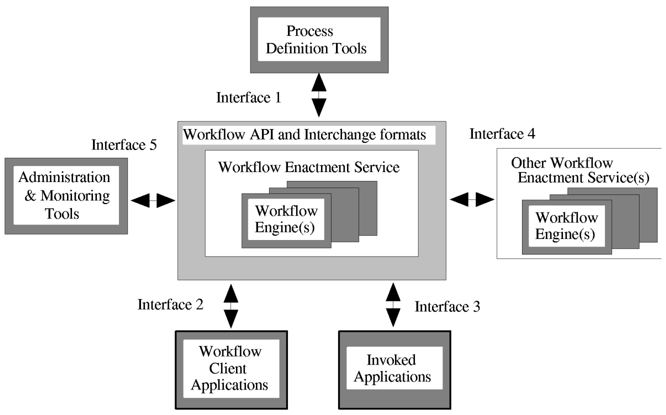

Process Definition Tools
Interface 1
Workflow API and Interchange formats
Interface 5
Administration & Monitoring Tools
Workflow Enactment Service
Workflow Engine(s)
Interface 4
Other Workflow Enactment Service(s)
Workflow Engine(s)
Interface 2
Interface 3
Workflow Client Applications
Invoked Applications

Figure 3.1: The workflow reference model
(WfMC 1995, p.20)

The core of the system is the workflow enactment service. It provides the run-time environment for one or more workflow engines. The workflow engine interprets process definitions received over interface 1. Humans interact with the workflow system through a client application that receives tasks over interface 2. Fully computerised activities are controlled via interface 3, invoked applications. Interface 5 provides access to third party administration and monitoring tools. Such tools may for example provide advanced statistical functions, or they may be used as a single interface to control several WfMSs in one place.

Multiple workflow engines are useful for distributing the workload among several processing nodes. Different workflow engines may also be customised for different application domains. They belong to the same workflow enactment service if they have a common naming and administrative scope, and they may interact and coordinate using vendor specific protocols. Interface 4, on the other hand, provides an API and exchange formats for interaction between heterogeneous WfMSs.

There are several reasons for combining heterogeneous components. Some tools may provide more advanced functions than what is included in a specific workflow product. Process definition tools may provide different modelling languages, analysis functions and simulation tools. An organisation may wish to build a client application that interacts with all their different WfMSs in a coherent fashion. And finally, a workflow may span different departments, e.g. sales and manufacturing, which use workflow systems adapted to their particular application environment.

3.1.1 Product Implementation Model

The product implementation model in figure 3.2 on the following page provides some additional details of the internal structure of the components presented above.

It identifies the main data elements being processed by a WfMS. An overview of these will be important in the following discussion on transactional features.

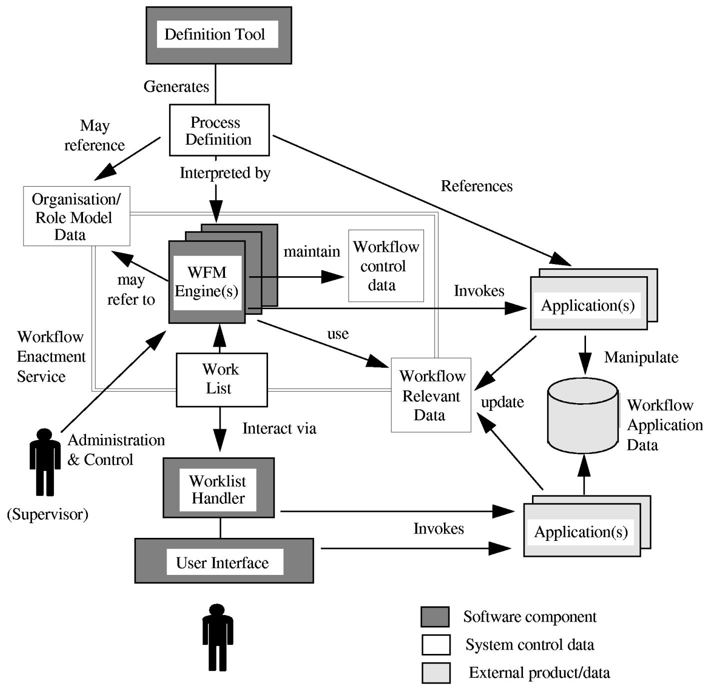


Figure 3.2: A generic WfMS product
(WfMC 1995, p.13)

The input to the workflow engine is the process definition and the organisation model. The process definition specifies the tasks to be executed for handling different types of cases. Tasks are inserted as work items in the work lists of selected participants. *Workflow participant* is a general term for human and computer resources that may process a work item. Human resources may be specified as an individual, a role, or an organisation unit. The workflow engine refers to the organisation model in order to select appropriate participants for the tasks.

The workflow enactment service maintains *workflow control data*. This data is internal to the system and identifies the state of individual process and activity instances (WfMC 1995, p.25). Data related to transactional control would fit within this category.

Workflow relevant data is used by the workflow engine to determine state transitions of a workflow process (WfMC 1995, p.26). This data may be accessed and updated by the workflow applications as well. *Workflow application data*, on the other hand, is only processed by the applications. It is considered to be inaccessible to the workflow engine, and references to it must be passed between tasks as unique object names or access paths.

From a transactional viewpoint this distinction between workflow relevant data and workflow application data may be problematic. Data under the control of the WfMS is “assumed to be” processed atomically, but the standards “does not include any specific controls over data synchronisation or recovery (for example between workflow execution, subflows or applications under execution)” (WfMC 1999*a*, p.37). In general, specific transaction mechanisms are left to the individual implementations, and it may be difficult to exercise sophisticated transactional control over external data while conforming to the standard at the same time.

3.2 Process Definition Meta Model

Workflow management systems are driven by high-level models of the business processes they support. A process definition is a “representation of a business process in a form which supports automated manipulation, such as modelling, or enactment by a workflow management system.” (WfMC 1999*b*, p.11). The process definition meta model defined in (WfMC 1999*a*) specifies the building blocks of a workflow, their relations and attributes.

Process definitions may be modelled in a graphical language. This is the most suitable format for human readability and will be the primary means of discussing workflow structures in this thesis. The graphical representation focuses on the routing of cases. It is typically expressed as a directed graph in which the nodes represent activities, and the edges represent the flow of work.

WfMC (1999*b*) identifies four types of routing between activities. They are illustrated in figure 3.3 on the next page. The exact graphical syntax varies between different languages, and even between different documents of the WfMC. The syntax chosen for this thesis is inspired by the informal presentation in (van der Aalst 1998).

Sequential routing: A segment of a workflow in which activities are executed in sequence under a single thread of execution.

Parallel routing: A segment of a workflow in which two or more activity instances may execute concurrently within the workflow. Parallel routing normally commences with an AND-split and concludes with an AND-join.

Conditional routing: A segment of the workflow in which there is a choice between alternative paths of execution. The branch is defined with an OR-split, and the alternative paths are merged with an OR-join.

Iterative routing: One, or more activities are repeated until a condition is met. The term *iterative routing* is taken from (van der Aalst 1998, p.7), while WfMC (1999*b*, p.34) defines this as *iteration*.

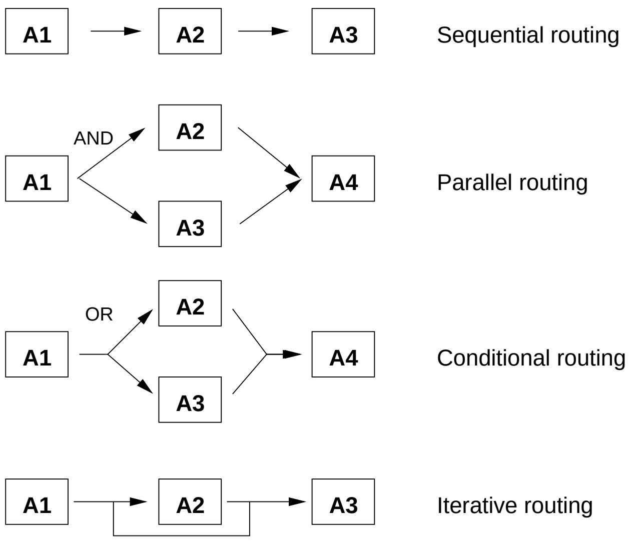


Figure 3.3: Graphical representation of routing constructs
(van der Aalst 1998, p.7)

Using these routing constructs, the process definition specifies the flow of work. Transition conditions may be used to block certain routes, but the default case is to make the transition between activities as soon as the from-activity has completed successfully. Apart from this, transitions are simple route assignment functions (WfMC 1999*a*, p.14). Conditions related to splits, joins and iteration are defined in the activities.

The activities represent a unit of work. They may assign a participant, referring to the organisation model, and applications to be executed, either automatically, or to support human users. Other attributes specify priority, routing conditions, and references to workflow relevant data to be processed.

The basic activity is atomic in the sense that it is a self-contained piece of work. Another type of activity is the subflow. It refers to a separately defined process, possibly to be distributed to another workflow engine. Some activities may be dummies that are included for routing purposes only.

The WfMC provides no way to specify transactional properties for the activities. Activities are considered to be atomic "with respect to the data under control of the Workflow engine" (WfMC 1999*a*, p.37), meaning that internal data must be rolled back or compensated in case of crashes or cancellation of the activity. However, the set of attributes for defining processes and activities is extensible (WfMC 1999*a*, p.21). Chapter 5.6 discusses extensions that will be useful for controlling the transactional mechanisms suggested in this thesis.

3.3 Alternative Workflow Definition Languages and Techniques

The Workflow Management Coalition is not the only organisation to provide workflow definition languages. In the area of web services, a there are many competitors to the WfMC. The Business Process Management Initiative seeks to standardise “the management of business processes that span multiple applications, corporate departments, and business partners”[2]. BPMI.org has released the BPML specification that allows for the definition of business processes that cross organisation boundaries. Microsoft and IBM have released the specification Business Process Execution Language for Web Services (BPEL4WS), which combines and replaces previous languages from both companies[3]. In addition there are a fair number of other alternatives for defining business processes.

In general, all the competing languages are able to express the four basic routing constructs described here. They differ in their approach to some advanced concepts (see van der Aalst 2003 for a comparison). For the most part, the conclusions in this thesis do not depend on which workflow language is chosen.

[2]http://www.bpmi.org/

[3]http://www.ibm.com/webservices/

Chapter 4

Case Study

This chapter describes LOVISA, a workflow system under development for the Norwegian Courts of Justice. LOVISA will support the preparation and documentation of criminal and civil cases, automating tasks where possible, and guiding the users in the flow of work appropriate for different types of cases. The first version of the system has been in production since 2003 and will gradually be improved and implemented in all the District Courts and the Courts of appeal within the next two years. I have done an analysis of LOVISA's transactional requirements and features.

The purpose of studying LOVISA as part of the work with this thesis has been partially to acquire knowledge for my own benefit, and partially to serve as an example when discussing general properties of WfMSs. The examples are used both as a means to clarify the discussion, but more importantly, they serve as a test to see how my general suggestions impact a concrete workflow system. Specific tasks in LOVISA have also inspired new ideas that might expand the usability of the Xymphonic Transaction Model.

van Leeuwen (1997, p.187) discusses pitfalls in designing workflow systems and states that "the automation expert must resist the temptation of focusing attention only on solutions to technical problems. Timely consultation with the real users is vital to ensure acceptance and suitability of the workflow." I have not discussed the design with potential users, but it is my aim to predict as far as possible which impact the suggestions proposed in this thesis will have on the users. Drawing on examples from LOVISA has been invaluable in this respect.

LOVISA is written in java using a framework developed by Computas AS called FrameSolutions. FrameSolutions provides classes and tools for defining and managing workflows and related data in an otherwise custom built information system. LOVISA is implemented in a three-tier architecture consisting of java clients running the user interface, an Enterprise JavaBeans application server that handles business logic, and a relational DBMS for storing persistent data.

LOVISA integrates with several external systems. The most important are Ephorte, a document management system for public bodies in Norway, and Microsoft Exchange server for scheduling the court hearings and other meetings in a case. Furthermore, criminal cases are received electronically from the police systems, and certain types of cases may be received from the Brønnøysund Registers, Norway's central register authority. Fees and other expenses connected to hearings are exported to an accounting system. Integration with external systems is common in WfMSs, and as such LOVISA is a representative example.

This chapter gives an overview of the architecture of LOVISA, and presents an example workflow that will be used throughout this thesis. The legal terms are translated from Norwegian by me using the dictionary (Chaffey & Walford 1997). Any errors are my own.

4.1 FrameSolutions

In order to understand the transactional features of the system, some background on FrameSolutions is necessary. The information presented here are taken from (Computas 2002), a programmers' guide to the framework.

4.1.1 The Workflow Meta-model

The workflow meta-model of FrameSolutions is in most respects compliant with the Workflow Management Coalition's (WfMC) workflow meta-model as presented in (WfMC 1999*a*). However, some variations in terminology warrant a description.

A workflow is defined as "The transfer of responsibility for a case from one organisation unit to the next, as they together perform the series of tasks needed to reach a goal" (Computas 2002, p.5). Basically, a workflow consists of tasks to be executed in some predefined order. Each task is assigned to one user by inserting it as a work item in his work list. It contains several steps to be performed by the user. The steps are defined in a process definition. The term process in FrameSolutions refers to a task, not a complete workflow as in much other literature.

A *step* is an activity, or a subprocess, which in turn may contain other steps. The activities of a process are the basic operations that the participant is to perform. Activities are typically very simple, e.g. answering a question, making a document, entering some data in a form, or creating new tasks. There may be constraints defining the order in which the steps are to be performed. Some steps may be hidden, depending on an evaluation of include conditions, and steps may be optional or required. Some steps may be performed repeatedly as often as desired.

The important difference between the reference model and FrameSolutions is the work item assigned to a workflow participant. WfMC (1995, p.19) states that "an activity typically generates one or more work items, which together constitute the task to be undertaken by the user (a workflow participant) within this activity". I.e. the WfMC see the work items as smaller units of work and the terminology states that a *task* is a synonym both for work item and for activity. FrameSolutions on the other hand, associates a process with the work item, thus a *work item* will contain several activities to be performed by the user.

To avoid confusion, I will stick to the terminology of FrameSolutions. Figure 4.1 illustrates a simple task containing some activities. It is defined in the process model, and the term process is often used as a synonym for task. Refer back to this figure as often as necessary. You may also want to take a look ahead at figure 4.6 on page 42 to see how different tasks are combined to compose a workflow.

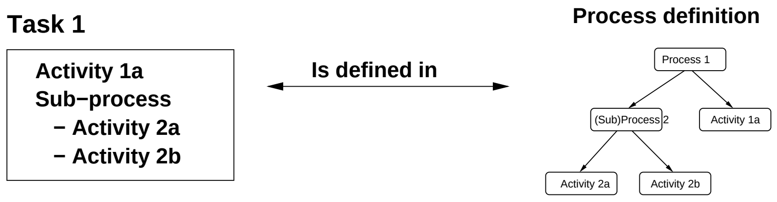


Figure 4.1: A task with some activities, and its corresponding process definition

The internal structure of an activity is illustrated in figure 4.2. When a user performs an activity, the system executes one or more actions. One action is typically one application statement, normally a java method invocation. Conditions control how the activity is to be performed. The activity may define pre- and postconditions as is also the case in the WfMC reference model. In addition, action-conditions allows a selection between two different sets of actions to be performed for the activity.

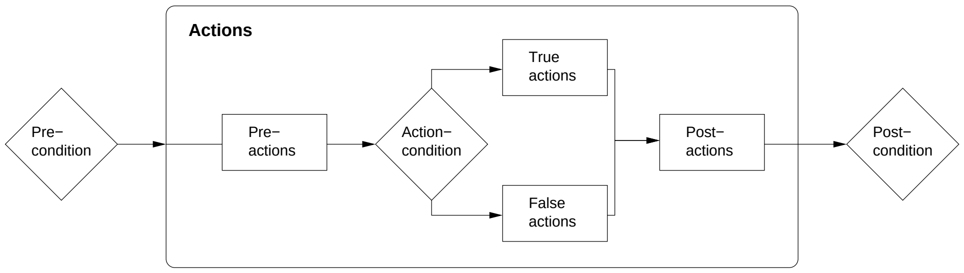


Figure 4.2: The internal structure of an activity. *A precondition must be true before the step can be started. An action condition is like an if-else-test in a programming language, selecting between two sets of actions to be performed for the activity. And a postcondition is the requirement for being able to terminate the step.*

Actions that interact with the user are specified to be run on the client. In such cases the workflow engine suspends the execution of the activity and a component in the client application is responsible for executing the action statement. The results of the action are returned to the application server upon completion, whereupon the workflow engine stores the data and resumes execution of the activity.

FrameSolutions applications handle both the workflow control data and the domain objects on which the applications operate. When a task is created, a process instance is created. The process instance is executable and it contains a state, referred to as context. Objects from the application domain model are bound to the variables of the context as work progress. E.g. a case is a domain object, representing the real world entity that is the focus of a given workflow. The case object is bound to the context variables of the first task of the workflow. It is passed along from one task to the next as work progresses, allowing all processes access to the case and the information that the case-object references.

To sum up, a process and the activities it contains are the building blocks of a workflow. A process is executed by one user when its corresponding task is selected in the work list. For work to flow from one user to the next, an activity in one process may initiate a new task that is placed in the work list of another user. Processes may also be created as a result of predefined triggers, e.g. the arrival of new information regarding a case, or the expiration of a deadline.

4.1.2 Transactions

FrameSolutions does not make the same distinction between workflow relevant data and workflow application data as does the WfMC. LOVISA manages both workflow relevant data and, to a large degree, workflow application data as objects in a domain model. The state of the objects in the domain model is stored in a central relational database together with workflow control data. Workflow control data consists of process instances created from the process definitions and other run-time information. Thus there are two types of data that are internal to the system – domain objects and workflow control data. Access to both types of data is controlled by the transaction mechanisms described in this section.

The exception to the above is data managed by external systems. The exchange of data with these systems is mediated by non-distributed transactions or by files in predefined paths in the file system.

LOVISA uses a form of multi-level transactions to access internal data. At the application server level, user transactions cover the processing of java objects. In general a user transaction lasts as long as the processing of one action[1]. Concurrency control is handled by a service in FrameSolutions called ObjectStore. ObjectStore sets read and write locks on java objects as they are read from the database. The database operation is a short read transaction. Execution of the action may initiate several new database transactions if it requests new objects to be displayed for

[1]*Action*: Recall that an action is one application statement, typically a method invocation, and that an activity consists of one or more actions

a user. When the action completes, the framework commits the user transaction by copying the new state of modified objects to the DBMS using a short update transaction, before releasing the locks in ObjectStore. Thus one user transaction at the application server level is implemented as several ACID transactions to the database.

The correctness of this approach is guaranteed by specifying that all write-access to the DBMS must go via the ObjectStore service. The code implementing an action must first acquire a write lock on an object before it can update its values. Read-only transactions may access the DBMS directly. LOVISA implements a number of search and reporting procedures by reading directly from the database. Also, individual cases may be opened and the objects displayed read-only, even though another user is currently editing the same data. In both cases, the object displayed is the most recently committed version. As soon as an object is updated, a new search, or a refresh on the client will retrieve its new values.

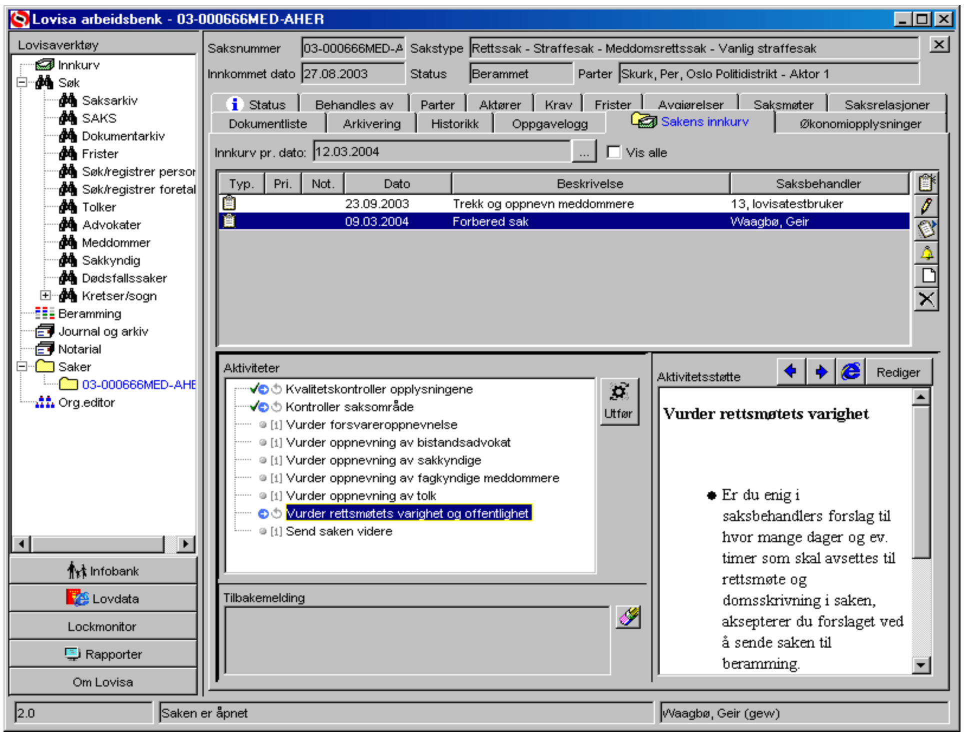
Objects are cached in memory for as long as possible to avoid the delay in reading from the database. Objects may also be cached on the client for quick access to its properties, and for processing of actions that are defined to be run on the client. The client returns the (possibly modified) object to the server upon completion of the action.

The LOVISA workflow enactment service is distributed over multiple application servers sharing the same DBMS. Each workflow engine serve one court district, and most of the data is processed by one district only. The exception is person information. People may be involved in several cases simultaneously and as a consequence, accesses to person objects must be globally controlled. The ObjectStore service includes a component that maintains locks on person objects in the shared database. The setting of a lock is protected by a database transaction that first checks for possible conflicts before persistently adding the new lock to the DBMS.

4.2 The User Interface of LOVISA

As the LOVISA client is started, the user's task list is displayed. It contains tasks that have been assigned to him for various cases, and there may be more than one pending task for a case. Upon selecting a task, the activities to be performed are displayed, and the user may execute an activity by selecting it and clicking an icon. Another option, when selecting a task, is to open the case folder. An example case folder is shown in figure 4.3 on the next page.

The case folder displays a *database view* of the developments in the case so far. The user may browse information such as the case's status, involved parties, legal claims, verdicts, documents, court hearings and other meetings, accounting information, history and case log (see the labels of the various tabs in the case folder in figure 4.3). By database view, I mean that the case is displayed without regards



Lovisa arbeidsbenk - 03-000666MED-AHER

Lovisaverktøy
- Innkurv
- Søk
  - Saksarkiv
  - SAKS
  - Dokumentarkiv
  - Frister
  - Søk/registrer persor
  - Søk/registrer foretal
  - Tolker
  - Advokater
  - Meddommer
  - Sakkyndig
  - Dødsfallssaker
  - Kretser/sogn
- Beramming
- Journal og arkiv
- Notarial
- Saker
  - 03-000666MED-AHE
- Org.editor

Infobank | Lovdata | Lockmonitor | Rapporter | Om Lovisa

Saksnummer	03-000666MED-A	Sakstype	Rettssak - Straffesak - Meddomsrettssak - Vanlig straffesak		
Innkommet dato	27.08.2003	Status	Berammet	Parter	Skurk, Per, Oslo Politidistrikt - Aktor 1

Status | Behandles av | Parter | Aktører | Krav | Frister | Avgjørelser | Saksmøter | Saksrelasjoner
Dokumentliste | Arkivering | Historikk | Oppgavelogg | Sakens innkurv | Økonomiopplysninger

Innkurv pr. dato: 12.03.2004 ☐ Vis alle

Typ.	Pri.	Not.	Dato	Beskrivelse	Saksbehandler
			23.09.2003	Trekk og oppnevn meddommere	13, lovisatestbruker
			09.03.2004	Forbered sak	Waagbø, Geir

Aktiviteter
- ✓ Kvalitetskontroller opplysningene
- ✓ Kontroller saksområde
- [1] Vurder forsvareroppnevnelse
- [1] Vurder oppnevning av bistandsadvokat
- [1] Vurder oppnevning av sakkyndige
- [1] Vurder oppnevning av fagkyndige meddommere
- [1] Vurder oppnevning av tolk
- Vurder rettsmøtets varighet og offentlighet
- [1] Send saken videre

Utfør

Tilbakemelding

Aktivitetsstøtte | Rediger

Vurder rettsmøtets varighet

- Er du enig i saksbehandlers forslag til hvor mange dager og ev. timer som skal avsettes til rettsmøte og domsskrivning i saken, aksepterer du forslaget ved å sende saken til beramming.

2.0 | Saken er åpnet | Waagbø, Geir (gew)

Figure 4.3: The user interface of LOVISA. *The left column gives access to the main functions of LOVISA. The most important are the inbox* (innkurv)*, several search functions* (Søk)*, journal and archive, and a folder of opened cases* (saker)*. An example case folder is depicted on the right. The fifteen tabs give access to different aspects of the case, for example status, involved parties (sorted under the headings* behandles av, parter, *and* aktører)*, legal claims* (krav)*, verdicts* (avgjørelser)*, documents* (dokumentliste)*, etc. The currently active inbox of the case* (sakens innkurv) *is displaying the two tasks that are pending. The activities of the highlighted task, prepare case* (forbered sak)*, is detailed in the lower half of the screen, including a text field explaining the current activity (the details are not important yet. The steps of this task are discussed in chapter 5.3.3). The activities are executed when the user clicks the execute button. Note the last activity, send case along* (send saken videre)*, which, when executed, will complete the task and create the next task of the workflow.*

to workflow management features. The user may even register and modify data in the case independently from the pending tasks in the task list. For example, in the tab displaying claims, the user may activate the same dialogue for registering new claims that he would get during the course of executing the task *prepare case*.

Support for adaptive workflows has received much attention in research the last five years. In LOVISA it is solved by a very simple mechanism. Figure 4.3 highlights the task list of the case. From this view, a user may start his assigned tasks, as well as other users' tasks, provided he has the rights to do so. Furthermore, he may delete and create tasks in an ad hoc manner to deal with unforeseen events. It is up to the user, as a responsible individual, to modify the workflow as appropriate to the circumstances.

These two properties, the database centric view of the case, and the options for ad hoc control of the workflow, leads me to the following conclusion: The workflow is guiding the users' actions rather than limiting, or enforcing them. This is a characteristic that will be useful for expert systems like LOVISA. It may be inappropriate for production workflows that manage highly formalised procedures. Thus, from this perspective, LOVISA is only representative for a certain type of workflow systems.

4.3 Example Workflow

The following is an overview of an example workflow from LOVISA. This is an introduction, and the relevant details will be provided as appropriate in the next chapter.

We will take a look at the workflow for criminal cases in which the accused has not admitted his guilt. This type of case is called *meddomsrett* in Norwegian, indicating that lay judges participate in the proceedings. These cases normally involve one professional judge, but if the prescribed penalty scale is above six years, two judges may participate.

A criminal case is initiated when the prosecuting authority sends an indictment and a summary of evidence to the courts, applying for a main hearing in the case. The receipt of the documents is registered in the journal and archive part of LOVISA. This creates the case object and initiates a workflow for the further processing of the case. The complete workflow is outlined in figure 4.4 on the next page. Note that the activities are omitted from this figure.

The triggering event is described in an oval and marked with a letter symbol to indicate the event was triggered by received mail. Each square is a task to be undertaken by the participant under whose column the task is placed (in the manner of UML activity diagrams). The arrows show the flow of work between tasks. Time proceeds downward. The goal is to complete a case within three months, however, in 2002 a significant number of the cases required more time. Other types of cases may normally take up to six months (*Yearly Statistics for the Courts of Justice* 2002, p.7).

The arrows show dependencies between directly related tasks. The precise semantics of an arrow, however, is that the source task will create the targeted task. Thus the dependency is implicit. There are also hidden dependencies. Tasks that seem to be endpoints have the semantics that they will just terminate when finished and not create any new tasks. Dependencies to other tasks' may be modelled by

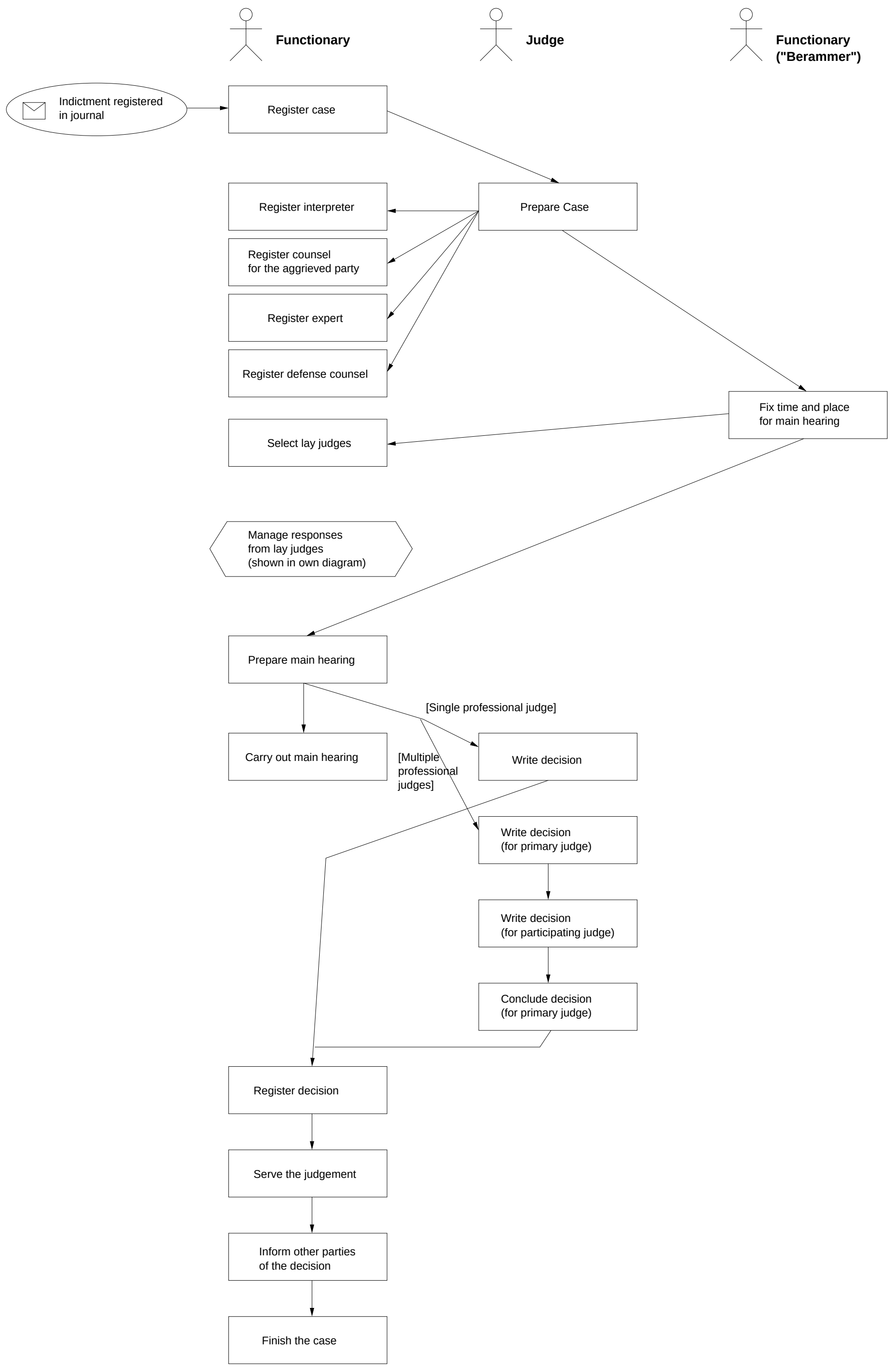

Functionary
Judge
Functionary ("Berammer")
Indictment registered in journal
Register case
Prepare Case
Register interpreter
Register counsel for the aggrieved party
Register expert
Register defense counsel
Fix time and place for main hearing
Select lay judges
Manage responses from lay judges (shown in own diagram)
Prepare main hearing
[Single professional judge]
Carry out main hearing
[Multiple professional judges]
Write decision
Write decision (for primary judge)
Write decision (for participating judge)
Conclude decision (for primary judge)
Register decision
Serve the judgement
Inform other parties of the decision
Finish the case

Figure 4.4: The workflow for criminal procedure

preconditions, but they are not shown. For example, the main hearing cannot commence until all the involved parties are registered and the required number of lay judges have confirmed participation (see the subworkflow in figure 4.5). Note that for the transactional features discussed in the next chapter, it is important to model all dependencies explicitly.

Some of the tasks create more than one follower. This is shown as an AND-split as discussed in chapter 3.2. For the most part, parallel execution is possible when the tasks are shown approximately side-by-side. Additionally, in the large court of Oslo (Oslo Tingrett), several functionaries with specialities in different fields may undertake tasks in parallel, although usually they will be performed one after the other by a single participant. Thus the workflow provides opportunities for parallel execution.

The hexagonal box represents a part of the workflow that is shown in figure 4.5. All the tasks in this subworkflow are started by triggering events, and ends without creating any new tasks. As noted above, however, there are dependencies not shown.

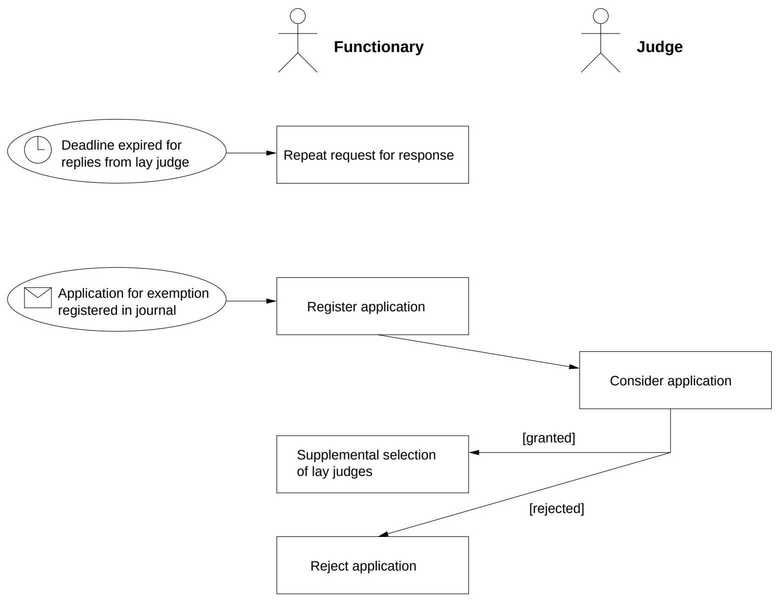

Functionary
Judge
Deadline expired for replies from lay judge
Repeat request for response
Application for exemption registered in journal
Register application
Consider application
[granted]
Supplemental selection of lay judges
[rejected]
Reject application

Figure 4.5: Manage responses from lay judges subworkflow

Each task in the workflow consists of several activities. As an example, figure 4.6 on the next page shows the detailed process structures for the tasks required to write the decision document. The exact procedure depends on how many judges there are for the case. I have chosen to show the branch where there are more than one professional judge. For simplicity, other tasks and arrows have been removed.

The notation used is the same as the users will see when they select a task in their task list. Required activities are marked with an inverted arrow. The circular arrow

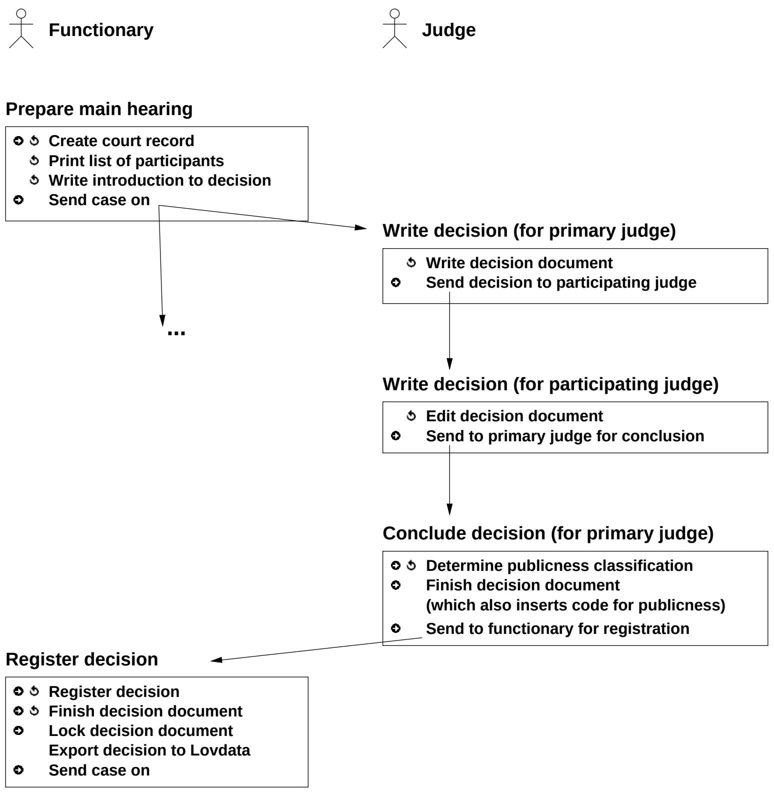

Functionary
Judge
Prepare main hearing
Create court record
Print list of participants
Write introduction to decision
Send case on
...
Write decision (for primary judge)
Write decision document
Send decision to participating judge
Write decision (for participating judge)
Edit decision document
Send to primary judge for conclusion
Conclude decision (for primary judge)
Determine publicness classification
Finish decision document
(which also inserts code for publicness)
Send to functionary for registration
Register decision
Register decision
Finish decision document
Lock decision document
Export decision to Lovdata
Send case on

Figure 4.6: Details of the tasks for writing the decision document

signifies that the activity may be repeated any number of times.

In general, the procedure is for a functionary to create the decision document in advance. He writes an introduction as part of preparing the court hearing. The document is temporarily stored in the database internal to LOVISA. It is sent to the primary judge (Norwegian: *domsskrivende dommer*), who will write his part, send the document to the next professional judge, who in his turn sends the document back to the primary judge.

When concluded, the decision is returned to the functionary for registration. The document is checked into the Ephorte archiving system by the *lock document* activity. A copy may also be exported to *Lovdata*, a legal information system containing, among other things, databases of verdicts.

A notable trait is that the writing is sequentially ordered. The document is passed between the editors in the context variables of the process instances. Another option using xymphonic transactions is explored in chapter 5.7.1. Also note that the transit between tasks is normally achieved by the user performing the last and required activity in a process. The workflow will not proceed until the user acknowledges the completion of his task by executing this activity.

All the tasks in the overview in figure 4.4 have a similar number of steps to be performed. Space does not allow all the details to be presented, but one more example is discussed in chapter 5.3.3 (see figure 5.11 on page 65).

4.4 Graphical Notation

A generic process (task), such as the one shown in figure 4.7, may have different internal structures, as exemplified in figure 4.8. For simplicity, I will assume that all steps are activities. A step may be a subprocess, but it is not executable in itself and will simply expand to a new set of activities. Usually the user will proceed from top to bottom in an orderly fashion, but as the diagrams show, he is often allowed to do this step here, another step there, and perhaps redo some steps before completing it all.

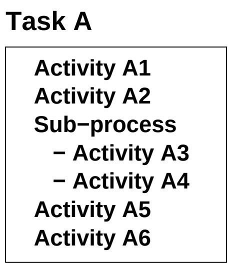
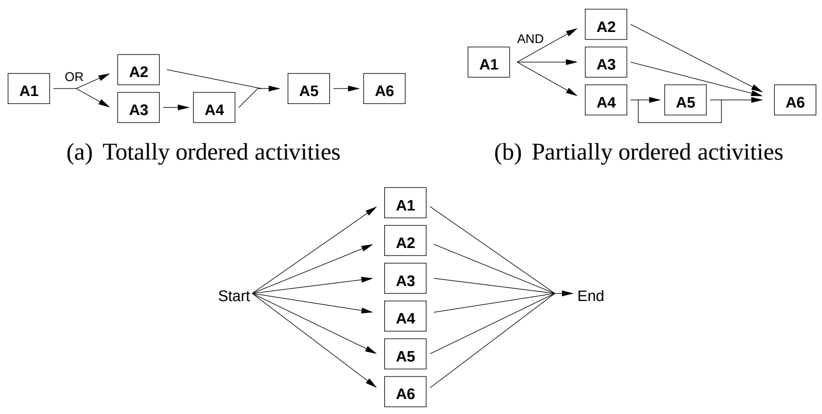


Figure 4.7: A generic task (process) in LOVISA



(a) Totally ordered activities (b) Partially ordered activities

(c) Unordered activities

Figure 4.8: Example activity structures

Figure 4.7 and 4.8 illustrate how the structure of a task, as it is defined in LOVISA, may be converted to the graphical representation proposed by the WfMC. This last form makes the internal structure more explicit and is therefore more suitable for the discussion of transactional properties. It will be used when discussing the internal structure of processes.

4.5 Summary

LOVISA is a workflow system developed for managing all kinds of civil and criminal cases in the Norwegian Courts of Justice. It will be used as an example to illustrate the discussion in the next chapter, and I will evaluate my suggestions on transactional design by showing how they would impact LOVISA.

This chapter started by describing the transactional design of LOVISA. Basically, there is one user transaction for each application statement in a process. For each user transaction, there may be several short transactions to the connected DBMS. The last part of the chapter gave an overview of the workflow for handling criminal cases. This workflow is relatively simple, still it includes all workflow elements that will be discussed in this thesis, including all the routing constructs, triggering events, opportunities for parallelism and integration with external systems.

It must be noted that the example presented only represents a small fraction of the workflows managed by LOVISA. Furthermore, LOVISA may have properties that are not found in other systems. I have attempted to focus on those aspects of the system that will be most representative for WfMSs in general. Hopefully the reader will be able to critically apply my ideas to dissimilar situations.

Chapter 5

Discussion

The Xymphonic Transaction Model controls concurrent access to shared data and includes some constructs for routing tasks between participants. However, it does not allow for the definition of the tasks to be done in a business procedure, and it does not in itself guide nor constrain the users in how they accomplish a task. In order to support and manage a workflow, process control will have to be implemented in a software component, which in turn may use transactions as a means for concurrency control and recovery.

In this chapter I propose a general model for how designers may specify the transactional behaviour of the processes they create, and how the workflow management system may interpret and enact this definition in terms of the transactions that are created.

The chapter starts with a discussion of architecture in section 5.1 and includes some design suggestions. Section 5.2 explores the upper bounds of transaction duration and introduces some useful terminology. The proposal for a mapping from workflow definitions to transactions in section 5.3 is the main contribution of this thesis. This mapping primarily establishes rules for structuring nested databases, while the uses for parameterised access modes are explored in section 5.4. Section 5.5 incorporates a scheme of data partitioning proposed in (Vaksvik 2002) for avoiding conflicts between workflows. The transactional design is made concrete in section 5.6 by presenting extensions to the WfMC's workflow definition language. Up to this point, the discussion focuses on supporting the behaviour of WfMSs as they are today. Section 5.7 explores some ways of extending the functionality of WfMSs. The discussion is rounded off in section 5.8 with a critical view on the atomicity requirement, and in section 5.9 some related work is presented.

5.1 Architectural Considerations

Transactions coordinate the storage of and access to data. The discussion starts by considering which data, from an architectural perspective, should be covered by the WfMS's transactions. These considerations will give an indication of which type of workflow systems may benefit from using xymphonic transactions. Following this, a design example based on Java Enterprise Edition (J2EE) is outlined.

5.1.1 Which Data should be Covered by Xymphonic Transactions?

In the reference architecture proposed by WfMC there is a clear distinction between the processing executed by the workflow enactment service and the data manipulated by the applications invoked during the workflow. Workflow application data is not used by the workflow enactment software and is relevant only to the applications or user tasks. It is normally placed outside the control of the workflow engine.

Such a distinction is probably inappropriate for a WfMS based on long lasting transactions. Consistency could be compromised if a distributed processing would fail in one part of the system, but succeed in another. Example scenarios include lost documents that according to the workflow system have been sent to customers, or vice versa, the documents may be saved, but the workflow history explaining the background for their existence may be lost.

Xymphonic transactions in particular would introduce problems if they were to integrate with non-xymphonic systems. The commitment of a data item to an external system done in a subxymphony might have to be revoked (undone or compensated) even though the subxymphony itself commits. No data committed in a hierarchy of nested transactions is finally committed until the top-level xymphony commits.

These possibilities for inconsistencies indicate that xymphonic transactions should be used throughout the system, that is, in all interactions between the workflow engine, the work list and the workflow client application. Data that is processed by invoked applications like word processors, spreadsheets and imaging programs, must be placed under transactional control by the workflow client. This may be achieved by checking out templates, or partially completed documents to the file system on the client computer, and checking in the results to the workflow system upon completion of the editing activity. Such a check-out check-in model is used for document handling in LOVISA.

The workflow reference model from the WfMC aims to allow a combination of workflow components from different vendors. However, it may be difficult to utilise xymphonic transactions effectively if mixing WfMSs. It is possible to support the definition of workflows in one tool as long as it can accommodate the extensions to the workflow definition language suggested in section 5.6. Interaction between different WfMSs may be handled as any other interaction with external systems. However, integration with a work list that does not support xymphonic transactions may be problematic. The presence of a task in the work list may be the result of an uncommitted transaction. The contents of the work list must be part of that same transaction in order to remove tasks in case of a rollback, and the client application should be built to interface with the non-standard transactional functions.

These considerations are supported by the conclusions reached by researchers in transactional workflows. Rusinkiewicz & Sheth (1995, p.600) state that "sometimes the data integrity constraints span the boundaries of individual databases and, as a consequence, the tasks accessing interrelated data must constitute an execution atomic unit." However, systems built for stand-alone operation normally do not provide the information and services that would be necessary to execute distributed transactions. Building a customised information system that manages most of the data homogeneously will provide advanced transactional support to most parts of a workflow, while limiting the number of interactions with external systems that constrain the applicability of advanced transaction models.

It must be noted that workflows, or parts of workflows, using xymphonic transactions can coexist in the same system with workflows based on ACID transactions. Anfindsen (1997, p.50) has shown that traditional serializability is a special case of the nested conditional conflict serializability property of xymphonic transactions. This means that both types of transactions can be serviced by the same transaction manager. Such a hybrid would allow for selected workflows, or subsets of tasks, to utilise the features discussed in the following sections, provided the subsystems on which they run can participate in the xymphonies.

5.1.2 Architectural Design Options

Custom built information systems are frequently programmed in an object oriented programming language using a relational DBMS for storing the objects. This is also the case for LOVISA, which uses the J2EE platform from Sun[1]. In this section I will discuss two design options for implementing xymphonic transactions in a system using an object oriented environment on application servers and clients. The suggested design is not the only way to do it, rather it is my intension to demonstrate the feasibility of incorporating xymphonic transactions in a WfMS.

Usually, such systems construct persistent objects from the database using a mapping to translate between the relational schema and the object model. The objects thus created are cached in the memory of the server and possibly even on the client. Modification to these objects apply to the cached copy, and are written back to the database according to the policy adapted by the object-oriented application. I will distinguish between two main approaches. The first option is to build support for xymphonic transactions into the workflow application and use traditional

[1]http://java.sun.com/j2ee

transactions to the database. The other option is to use xymphonic support in the DBMS, keeping the database connection open as long as there is a live transaction holding locks on the corresponding object, and possibly writing uncommitted modifications back to the database several times during a transaction.

Implementing Xymphonic Transactions in the Object-Oriented Application Server

Taking LOVISA as an example, xymphonic transactions may be built as an extension to ObjectStore, the service that coordinates write-access to objects cached in memory. In LOVISA, ObjectStore is a fairly simple lock manager, which all requests for objects in the application server must pass through. It could be extended with more advanced transaction manager services to support xymphonic transactions.

First, we will get an overview of the various cached copies of an object. The system may contain up to three versions of an object x. As shown in figure 5.1 these possible versions are:

- The most recently committed value x_0
- The most recent version x_1 of an object that has been committed in a subtransaction, but is not finally committed, and
- The object x_2 that is in the process of being updated by a live transaction.

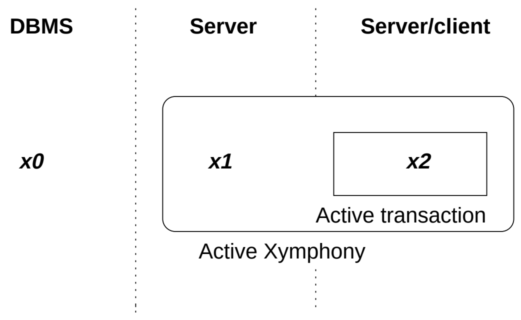


Figure 5.1: Versions of an object x

The version x_0 will reside in the DBMS and may be retrieved by issuing SQL-statements directly to the database. The object x_1 is cached and associated with the committed state of a xymphony. It may be accessed in write-mode by subtransactions of the xymphony or any of its children. It may also be accessed in read-mode by any transaction using a parameterised read, and whose parameter set is compatible with the parameter set of the xymphony. Requests for x in write-mode will create a copy x_2 of x_1 that may be processed in an activity, either on the server or on the client. Upon completion of the activity, x_2 will replace x_1, and upon commitment of the xymphony, the current state of x_1 should be written to the database using a short ACID transaction.

This design makes rollback of a xymphony a matter of simply discarding all its cached objects. Rolling back a subtransaction amounts to discarding the transaction's x_2 objects. However, in order to support rollback of subxymphonies, ObjectStore would need a transaction log recording the before-images (which is commonly used in the undo-redo protocol described in Bernstein et al. 1987, pp.180–195) of all x_1-objects that were replaced in a subxymphony. And finally, the lock manager of ObjectStore would have to implement parameterised lock modes and the mechanisms needed for the inheritance of locks downwards and upwards in the nested xymphonies.

The benefits of implementing xymphonic transactions in the application server are that the database transactions may be kept short. Not all databases support long-lasting transactions, and the database performance would not suffer from handling a large pool of live transaction state and locks. The performance issue, however, would move to the application server. Cached objects would have to be kept in memory for a longer time, unless a scheme was devised to allow the objects to be temporarily written to disk (e.g. by adapting the complete undo-redo protocol to the case of cached data). The ObjectStore lock manager would also have to keep an increased number of locks in memory, and each lock would be more complex.

The performance issues can be alleviated to some degree by introducing more application servers. LOVISA already has this option. The database server on the other hand cannot that easily be partitioned and distributed on more nodes. Thus it may be easier to handle the performance issues in the application server, than in a central DBMS.

Needless to say, a drawback of this approach is that the implementation would require the programming of an almost complete transaction manager. As commented by Gray & Reuter (1993, p.485), this is a great task and the debugging may take years. Thus the quickest approach would be to exploit the transaction mechanisms already available in a commercial product.

Using Xymphonic Transaction Services in the DBMS

This option should be fairly easily available by using the existing Xymphonic Engine, or possibly in the future, its equivalent. The DBMS would handle all transaction mechanisms, including concurrency control, recovery and the management of xymphonies. The application server would have to maintain an open connection to the database for the duration of the whole xymphony. There should be a one-to-one relation between the user transaction and the database transaction.

In this case, the commitment of a subtransaction on the level of the application server should probably be propagated down to the DBMS. Depending on the buffer management policy, the previously committed version x_0 of an object would be written to the log and replaced by the committed state of x_1. Thus access to any version of x would be restricted to parameterised read operations compatible with the write-locks for the entire duration of the top-level xymphony.

Transaction management on the server would be simplified. It would have to provide its clients an interface to the transaction mechanisms available from the DBMS. It should probably not keep its own locks, as they would be redundant and difficult to synchronise with the locks managed by the database. Rollback would be managed by the DBMS, but should be complemented by the application discarding all the cached objects covered by the aborting (sub)xymphony or transaction.

It is quite possible that allowing long-running transactions in the current DBMSs would significantly reduce the number of clients that they are able to serve. The experience of the LOVISA developers is that when using a maximum duration of 30 seconds for a database transaction, the performance is marginally acceptable. The transactions suggested in this thesis could easily last for more than an hour. Apart from these words of warning, however, I will merely assume that performance will not be a problem.

5.1.3 Who is the Owner of the Xymphonies?

A hierarchy of xymphonies is ideal for delegating work. The one who starts a task is the owner of the top-level xymphony and may delegate parts of the work to other participants. He holds responsibility for the final outcome of the work, and this is supported technically by giving him the rights to determine which subtransactions to accept and reject.

A workflow, on the other hand, is typically initiated by a trigger such as an incoming letter, telephone, order, etc. The recipient may be a secretary whose task it is to register the event in the system, and the workflow may proceed to several participants who has authority in their respective areas/fields. Giving ownership to the user who starts the workflow is therefore inappropriate.

The only proper solution is to give the workflow engine ownership and full authority to the xymphonies. The engine can then check the user's rights to manage transactions against the organisation model. In this way, configuration of access rights to transactional features can be integrated with the definition and maintenance of other aspects of the user organisation. However, the question of how to assign user privileges will only be briefly touched in the following sections. A detailed discussion of this topic is outside the scope of this thesis.

5.1.4 Summary of Architecture

WfMSs designed for a heterogeneous environment seems to be unsuitable candidates for incorporating xymphonic transactions. A workflow in this setting is potentially distributed among nodes with limited transactional support. Inconsistencies could be the result of diverging transactional semantics for different activities.

Xymphonic transactions are more suitable for implementation in a homogeneous system. As a minimum, both the workflow engine, the work list and the workflow client application should be able to participate in a distributed xymphonic transaction. Interaction with external systems is possible, but will reduce the potential transaction duration. This last point is further discussed in the next section.

5.2 Upper Bounds on Transaction Duration – How Long May a Transaction Last?

Frame Solutions and LOVISA uses one transaction to a task, and the tasks are designed so that they should be as short as possible. One benefit of the Xymphonic Transaction Model is that it allows for long-lasting transactions. The question is, how long may the transaction last?

One aspect of this question is how to exploit the Xymphonic Transaction Model in order to allow shared access to data by long running concurrent transactions. This is the topic of the better part of this chapter. However, we will start by looking at the factors that may reduce transaction duration. Some of these result in absolute constraints that we cannot hope to avoid using the technology available today.

The following factors may limit the maximum duration of the transactions:

- Avoiding lost work
- Real actions and the requirement for auditing and accountability
- Integration with external systems

A transaction lasting for a day or more has a greater probability of being interrupted by system crashes, or stops due to maintenance. Atomicity requires that all active transactions must be aborted when the system is restarted. The problem is that a lot of work may be lost when the transaction is forced to abort. Transactions may be kept short in order to avoid lost work. A more elegant solution would be to use persistent savepoints, which are discussed in section 5.2.1.

Real actions were discussed by Gray & Reuter (1993, p.163). Once executed, they cannot be undone. For auditing purposes, the data regarding a real action must often be recorded durably in the database. This requires the transaction to postpone the execution of the real action until it is prepared to commit, and upon completion of the real action, it must finally commit. The implication is that a task involving a real action forces the transaction to commit. The presence of real actions effectively prevents transactions that cover the whole workflow. Its relevance to WfMSs is discussed in section 5.2.2.

Workflow systems are frequently integrated with external systems. The external systems may not be able to run long-lasting transactions. In order to preserve consistency between the WfMS and the external system, the transactions that are durably committed in one system should also be durable in the other. The workflow cannot be undone, unless it is possible to roll back, or compensate its effects in the external systems. Unless compensating transactions are available, we will have to treat interactions with external systems as real actions. This is also discussed in section 5.2.2.

5.2.1 Persistent Savepoints to Avoid Lost Work

In a system designed for long-lasting transactions it is desirable to provide forward recovery in order to limit lost work due to crashes. Persistent savepoints have been proposed to allow restoration of the data to the last savepoint. Gray & Reuter (1993, p.575) state that to their knowledge, no system had at that time implemented persistent savepoints. The Xymphonic Engine provides persistent savepoints, and it is probable that more recent systems do as well.

The availability of persistent savepoints is important to determine upper limits on transaction duration. An organisation might decide that it will not risk loosing more than a few hours of work. Unless persistent savepoints can be used to save the current state of the workflow every hour or so, this may force the transactions to commit frequently.

For forward recovery to a persistent savepoint to be consistent, it is necessary to restore both the data that is processed by an application, and the application context as well (Gray & Reuter 1993). Some of the application contexts that typically will be present in a workflow system are outlined in the following.

Recovery of Application Context

In a typical office environment many operations on a workstation will not be protected by transactions. The data processed by word-processors, spreadsheets, email clients and web browsers may be protected by transactional OSes, but the application context (e.g. local main memory and temporary files) is usually unprotected.

Thus the data involved in a work procedure may be partially protected e.g. in a customer database, and partially unprotected in an unsaved document. Cut and paste operations between such programs may compromise isolation, and crashes may lead to inconsistency between the unprotected application state and the data that is protected (and possibly persistent).

This implies that a persistent savepoint (or a commit) should be taken in such a way that the application context is recoverable as well. Word-processors and spreadsheets must save the document before a savepoint is taken. An email client must send all messages or save them as drafts. Some data may not be saved. For example window positions, and browser history is usually only saved on program exit. These are examples of state that may not be that important, and its possible loss may be ignored. However, a persistent savepoint should only be taken if all state that is relevant to the application logic can be included in the savepoint.

5.2.2 Real Actions and Auditing

One of the motives of Vaksvik (2002) was a to allow an undo-operation when executing a workflow. This seems to be easily achieved by executing the tasks in long-running transactions, partitioned by subtransactions or savepoints. An undo of a task amounts to aborting the subtransaction that contains its work, or rolling back to an appropriate savepoint.

This approach is not novel, and the following argument demonstrates that it may not always be possible.

> Rollback in a transactional sense is not applicable in case of long running applications like workflows. However, there is a strong request for a feature, which allows to execute "something" which undoes some or all the work of a process (e.g. to add a "not longer valid" phrase). In addition, it is a request that these "undo executions" can be distinguished from work which is done to reach the goal of the process. Especially in applications of the public administration, it is a requirement to be able to "cancel" a request. On the other hand, it is totally forbidden that documents of processes are just destroyed (i.e. no rollback in a transactional sense). Instead, an additional document has to be prepared which invalidates the process.

(Schwenkreis 1996)

The problem is related to the concept of *real actions* (Gray & Reuter 1993, p.163). A protected action is an operation that has the ACID properties. Undoing its effects will not affect the outside world. A real action, in contrast, cannot be undone, as is the case for most real world actions. The primary example is drilling a hole or in other ways demolishing physical entities. Examples from the area of workflows may include sending a letter, acknowledging the receipt of a document, or committing a transaction in a legacy system. It would be inappropriate for a workflow system to undo the information regarding such actions.

A transaction involving a real action must delay the action itself until the transaction is guaranteed to commit. Such a guarantee is not possible to give if the transaction should proceed doing updates. Even a nested transaction that successfully commits cannot be guaranteed to finally commit. This means that a task requiring durable records of its results will force the whole xymphony that it is a part of to commit.

A workaround would be possible if we considered the log to be a durable record of the real action. However, the log is not the database. Any updates in the log that belongs to aborted transactions are invalid. Due to the atomicity requirement, such updates have not happened.

Real actions are initiated by, or through the system. Recording data about other real world activities may also be required for auditing purposes, although the WfMS is not involved in the activity itself. The WfMC (1999*a*, p.33) allows the definition of manual activities in a workflow process. The WfMS will inform a participant about the task to be undertaken, and it expects a response as to the outcome. Bowers, Button & Sharrock (1995) describe a workflow system that was introduced in a print shop to account for the progress of print jobs for a major contractor. On a panel in the print shop floor, the operator would punch in a job number to inform the system of the starting and finishing times of the job. In this example, it would be unacceptable for the transaction recording the job operation to abort. The operator could manually enter the jobs at the end of the workday, but to do it twice, because of some system peculiarity, would be intolerable.

The updating of external autonomous systems is another important class of real actions. One of the strengths of WfMSs is their ability to integrate data processing on disparate systems, often legacy systems. The problem is often that such systems will not participate in a distributed commit protocol like 2PC. Some systems may not even be transactional in nature. The systems will seldom be able to hold their commitment for any extended length of time. Thus the update of an external system will be short and commit immediately. For auditing purposes, such an update will probably force the workflow transaction to commit as well.

Compensatable Tasks and Pivots

In order to treat the different types of real actions in a precise way, I will use the terminology introduced in the literature about flexible transactions (Elmagarmid, Leu, Litwin & Rusinkiewicz 1990, Leu, Elmagarmid & Boudriga 1992, Hagen & Alonso 2000, Schuldt, Alonso, Beeri & Schek 2002).

> Informally, a compensatable task is one that can be undone one way or another in case the process either fails or it is cancelled. In most real world processes, compensation is possible by executing a number of actions that cancel the effects of the initial tasks. These actions may

> be directly related to the task (e.g. a transactional undo) or be a semantic compensation (e.g. a letter is sent notifying the user of a given mistake). By labelling a step as compensatable we are acknowledging this fact. On the other hand, a task is a pivot when the overhead or cost of compensating it is not acceptable (note that the definitions are not mutually exclusive, it depends on the concrete application). Committing a pivot task means we are committed to complete the process because, otherwise, things will get expensive or difficult.
>
> (Hagen & Alonso 2000, p.946)

Designing compensating transactions is difficult, so they are assumed not be available as the general mechanism for rollback. But any transaction, or xymphony, is in effect compensatable until it commits. Thus we may use the above terminology to describe workflows using xymphonic transactions. The tasks that for some reason must commit are termed pivots. Any other task is compensatable as long as it can be aborted (i.e. its xymphony has not yet committed).

When a pivot activity completes, the workflow engine must prepare to commit the top-level xymphony and all contained subtransactions. When all participating subsystems have voted to commit, the real action may be executed. As noted in (Leu et al. 1992, p.173), commitment of the non-compensatable subtransactions must wait for a global decision. In order to avoid waiting for concurrent tasks that may take a long time before they can be prepared to commit, we should avoid running pivots in parallel with other tasks. Section 5.3.6 discusses how to handle this.

Note that we will define more activities to be pivots, than what is the case if compensation is possible. For example, when committing a transaction in a legacy system, the activity is regarded as a pivot in order to ensure that consistency is preserved. If sending a letter to a customer, it is important that the organisation does not risk loosing their copy of it. However, these are actions that in some situations may be compensatable, and if compensating transactions are available, they are not regarded as pivots.

In (Schuldt et al. 2002) the pivot has another important meaning. Schuldt et al. require that a process program includes at least one execution path following the pivot that is guaranteed to terminate the workflow successfully and consistently. Such a requirement is not adopted in this thesis.

5.2.3 Summary of Duration

A pivot is a real action, or any task whose processing is required to be durably recorded. A pivot cannot be compensated, and it must finally commit upon successful execution. This forces the transaction and all parent xymphonies to commit.

If the database cannot provide persistent savepoints that may also recover application state, the desire to avoid lost work will further constrain the maximum duration of a transaction. The organisation policy will probably determine the appropriate time-limits.

Having discussed the factors that require a transaction to terminate, we now turn to a consideration of how to accommodate long-lasting transactions.

5.3 Mapping from the Process Definition to Transactions

The workflow definition is the input that specifies the tasks and activities that must be performed in a case. This section discusses general rules for how the workflow engine may interpret the workflow definition in order to establish appropriate transactions. By applying these rules, we will be able to map from activities to transactions and xymphonies as the workflow proceeds.

Today's WfMSs support process change by merely changing the process definition (van der Aalst 1998, p.4). Such changes, especially if made after deployment of the system, are possibly performed by inexperienced developers. In particular, if developers are unfamiliar with xymphonic transactions, it is desirable to provide consistent guidelines for transactional design. It would be even better if the workflow engine was able to establish suitable transactions automatically. It is my aim to come as close to this goal as possible, and at least simplify the design issues facing developers.

The basic rules are outlined in section 5.3.2, which introduces an advanced undo functionality for the tasks of a single user. Next we will see what happens when this design is applied in the context of multiple users. The four routing constructs sequential, conditional, iterative and parallel routing affect transactions in different ways. Sections 5.3.3 through 5.3.6 focus on each of these respectively.

5.3.1 Preliminary Remarks

Activities that require read-only access to its data are rare in LOVISA. Most activities amount to registering some fact about the case. I don't know if this is typical of WfMSs, but assuming it is, all concurrent activities that process the same data will introduce write-write conflicts unless they are run within the same xymphony. For this reason, my design is directed towards supporting the structure of the workflow with a corresponding structure of nested databases.

Parameterised read operations, on the other hand, allow read-access to objects that are being updated by another user. This will come in handy when transaction duration increases. The uses for parameterised access modes will be discussed in section 5.4.

Dependencies will exist between the tasks of a process definition. The most common use is to define an execution order. This type of dependency is the focus of the graphical representation of a business procedure. Reijers (2003, p.9) notes that "dependencies may have various other semantics, expressing for instance an information exchange or control dependency." Dependencies related to which common data objects will be processed by multiple activities are very important to transaction design decisions.

In the following discussion, I will assume that an order defined over some activities implies a dependency from the results of one activity to the following. This means that an activity cannot be rolled back unless all the following activities also have been aborted. On the other hand, I assume that the absence of a defined order implies that there is no dependency between a pair of tasks, and in fact that the two tasks process disjoint sets of data. This last assumption may be untrue in LOVISA, but that should be possible to correct. For each pair of tasks that need write-access to some overlapping set of data, we must either supply an ordering, or introduce a property that tells the system to run the pair in the same subdatabase, regardless of their relative order. We may indicate such a dependency by drawing a two-way arrow between the activities (or an n-way arrow if n unordered activities process overlapping data). The graphical representation of a two-way dependency is shown in figure 5.2.

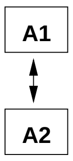


Figure 5.2: A two-way arrow signifies unordered activities with a dependency

5.3.2 Single-User Mini-Workflows

Some tasks in a workflow consist of a set of steps to be executed by a single user. In LOVISA this is in fact the definition of a task. The steps, or activities, defined in the corresponding process definition, may be unordered, partially ordered, or totally ordered. This allows the task to be depicted as a mini-workflow that is the responsibility of a single user.

Logically, a process in LOVISA is a unit that probably should have the ACID properties. Having xymphonic transactions, it might be even better to give it the ACCID properties, allowing parameterised read-access during processing. Further, a process is an ideal scope for undo-functionality. A user proceeds from one step to the next, and may at some time decide that a previous decision was wrong. Using flat transactions with savepoints, he would have to undo, in a reverse order, all the steps leading back to the error. But using nested transactions, or nested databases, he may be allowed to undo selected tasks in isolation.

The following discussion assumes that no activity, except for the last, will create a task for another user. Such activities are in effect an AND-split, and will be treated in section 5.3.6.

Proposed Design

I propose the following rules for nesting transactions within the context of a single user's task:

Start: Create a xymphony, or a subxymphony for the task. This will be referred to as a task-xymphony, indicating that it is the parent of all the subtransactions of this task.

Parallel routing: Create a subxymphony for each of the activities following an AND-split. If more than one activity is a starting point of the process, treat this as if there existed a dummy activity with an AND-split leading to the start-activities.

The exception to this rule is activities that have the data-dependency property. All activities having an n-way arrow between them should be run in the same subxymphony and otherwise treated as a sequence.

Commit the parent xymphonies of parallel paths before executing the activity following their AND-join. At this point, the previous activities cannot be aborted individually. Establish a new subxymphony for the activities following the and-join. This makes abort of the following activities possible without affecting the committed activities preceding the AND-join.

Figure 5.3 on the following page sums up the rules for parallel routing.

Sequential routing: The whole sequence will run as a single transaction with savepoints. Establish a savepoint at the end of each completed activity. Commit the transaction when the chain reaches an AND-join or an AND-split. See below for a justification for using savepoints rather than nested transactions.

Activities with the n-way dependencies will be treated as a sequence. This of course assumes that the user is not allowed to start several activities simultaneously. The user will perform the unordered activities in a pattern that is unknown before it actually happens, still it will follow a single sequential thread of execution that may be divided by savepoints as suggested here.

Conditional routing: Treat the activity instances as a sequence. The actual execution will proceed sequentially along a single path of execution. This corresponds exactly to the trip planning example presented in (Gray & Reuter

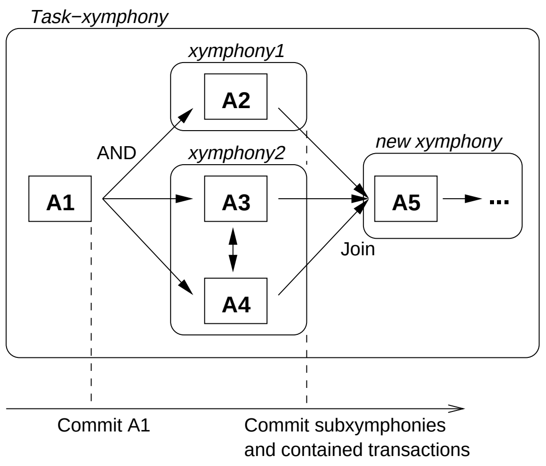


Figure 5.3: The xymphonies resulting from an AND-split

1993, p.171), which shows how savepoints may be used to tentatively book a flight route through the world. If an obstacle is encountered, the transaction can backtrack to a savepoint and try another path.

Iterative routing: This is a special case of sequential routing. Activities that may be repeated add consecutive activity instances to the sequence.

Figure 5.4 sums up the rules for sequential routing, including examples of OR-splits and iteration.

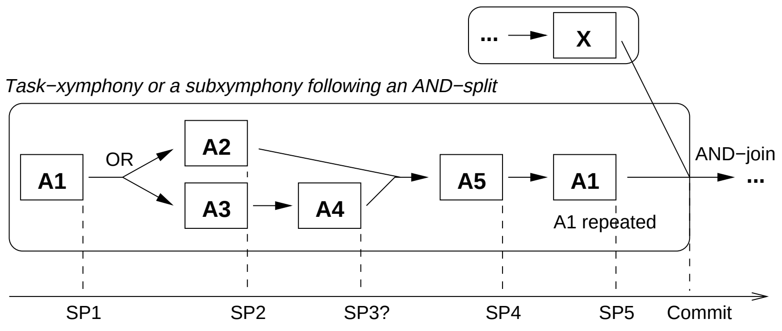


Figure 5.4: Using savepoints (SP) for controlling sequential activities

Pivot: Whenever a pivot forces the top-level xymphony to commit, establish a new xymphony for the remainder of the activities to run in. All the rules are followed as if the process was newly started, e.g. if there are multiple activities from which to continue, this is treated as the multiple entry points discussed above.

The key elements are the AND-splits and AND-joins. By establishing a xymphony for each of the activities following an AND-split, these may be undone individually regardless of the order in which they were executed. This is consistent as long as there is no data-dependence between the tasks.

The AND-join presents a challenge to transaction processing. The xymphonies of the joined activities must be merged, but ideally, if the AND-join is aborted, they should be restored, again allowing selective rollback of the individual activities. A transaction system allowing such behaviour would be a step closer to realising the spheres of control suggested by Davies and Bjork (Gray & Reuter 1993, pp.174–180). Being limited to nested databases (and the same holds for nested transactions), the xymphonies must be "merged" by committing to their parent database. There is no way this commitment may be cancelled without breaking atomicity.

Thus the activities A2 to A4 in figure 5.3 on the preceding page may be aborted singularly until the AND-join, at which time they may only be aborted as part of a rollback of the task-xymphony.

Design Options for Sequential Routing

In the case of a sequence, however, there is a dependency between the activities. This may result from the activities processing the same data, successively refining the result, or because the result of an activity is an input to the following task. This requires a rollback to proceed in the reverse order back to the point selected by the user.

I will consider two options for implementing this behaviour with transactions. We may use nested databases, nesting each consecutive activity within a new subdatabase. This option is shown in figure 5.5. Each activity commits its transaction when finished. Its locks are inherited by the parent xymphony and may be acquired by subtransactions. The next activity establishes a subxymphony within the parent. Upon completion of the sequence, the hierarchy of xymphonies may commit. A rollback is accomplished by aborting the subxymphonies as far back as desired.

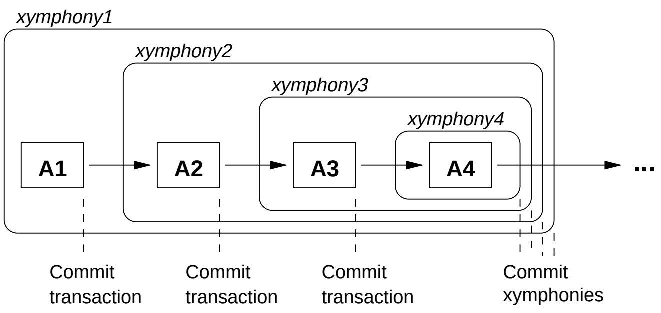


Figure 5.5: Using nesting for sequential activities

The other option is to use savepoints as shown in figure 5.6. For each finished activity, set a savepoint instead of committing. Commit the sequence when a rollback is no longer needed or desirable. In practise, there may be even more savepoints than shown. An activity will probably have one savepoint for each transaction that otherwise would have been used, e.g. for each action of an activity.

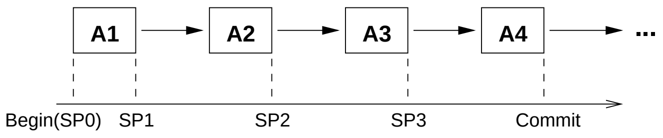


Figure 5.6: Using savepoints (SP) for sequential activities

I prefer using savepoints for the simple reason that they consume fewer resources. Nested databases would accommodate a flow between multiple users, but within the context of a mini-workflow this is not needed. Both allow for backwards recovery, but savepoints can be easily set within different parts of a transaction, possibly making the scope of rollback even smaller than an activity. Using savepoints, however, requires the suspension of the transaction after completing each of the sequential activities and possibly executing another activity before the transaction is resumed. Current database systems may not allow the clients to switch between different transactions in this manner. In that case, using xymphonies for sequential activities is the only option, and the rules must be modified accordingly.

An Example Task from LOVISA

Following the creation of a criminal case, a functionary does some preparatory registration and classification. The task *register case* is the first task in the workflow. Its detailed specification is shown in figure 5.7.

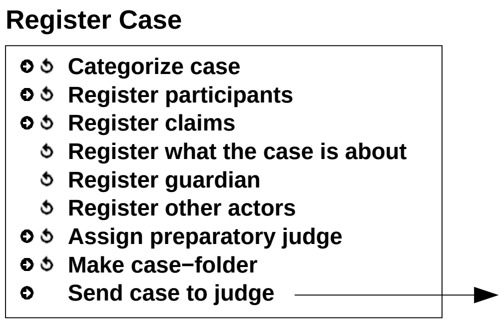


Figure 5.7: Example task from LOVISA

There is no order defined for the activities, except for *send case to judge*, which must be executed last. Thus the routing diagram in figure 5.8 on the following page is quite simple: There is a dummy start activity with an AND-split to the first eight activities, which is joined in the last activity.

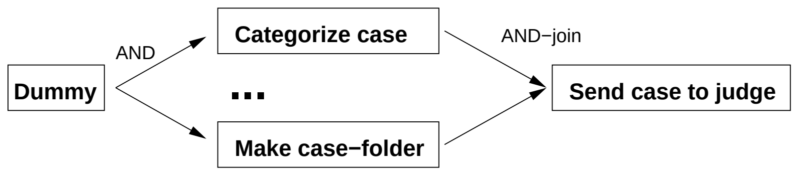


Figure 5.8: Internal task structure of register case

Using the transactions proposed in this section, the system would establish a subxymphony for each of the first activities. After completion of an activity, the transaction would set a savepoint and then suspend, allowing for a repetition of the activity. Regardless of the order in which the activities were done, any activity could be undone by aborting its subxymphony. When the user decides to send the case to the judge, and the system has verified that this activity is allowed to run, all active transactions and subxymphonies would first be committed. After this, undo would not be permitted for the individual activities, but this is not a problem, since no more work will be done with the task.

A Complication

Redoable activities complicate the clean scenario drawn so far. If an activity preceding an AND-split is redoable, there will be dependencies leading from this activity a second time. The logical approach is to say that whatever activities were executed after activity A1's first invocation should be committed before A1 is executed a second time. The second invocation should produce a new subxymphony to contain this second execution as if it followed an AND-join. An example scenario is diagrammed in figure 5.9 on the next page, in which activity A2 and A3 were completed after the first invocation of A1, and activities A4 and A5 are executed after the second invocation.

The illogical thing about this situation is that activities A4 and A5 may be aborted as usual, whereas A2 and A3 can only be aborted as part of undoing the whole task. The user may be accustomed to the option of undoing executed activities, but suddenly it becomes unavailable for some of them. We might be hard pressed to explain this seemingly inconsistent behaviour to a user, and he might not be willing to abort the whole task. Furthermore we have a problem in deciding how to run an activity A6 following e.g. A2. According to our rules, this should be run in the same xymphony as A2, preferably in the same transaction, following a savepoint. But since the transaction of A2 is committed, the only option is to run A6 in a new subxymphony.

A solution to this problem is to disallow the redoable property for activities that

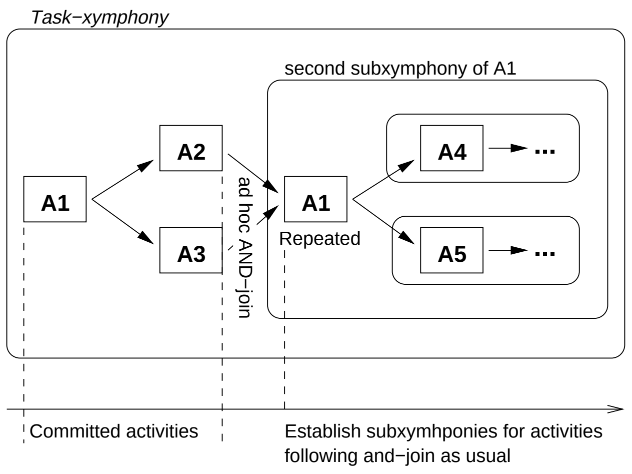


Figure 5.9: Default behaviour for redoing activities preceding an AND-split

precede an AND-split. This may constrain the flexibility of the process model to such degree that one might be better off not using long-lasting transactions. Another option is to allow the redoing of activity A1 without committing the already established subxymphonies of A2 and A3. However, this may lead to write-conflicts between the second transaction of A1 running within the task-xymphony and locks held by the subxymphonies of A2 and A3.

A similar constraint on the undo functionality arises when committing a pivot. As discussed earlier, a pivot forces the top-level transaction to commit. This effectively prevents all possibilities of aborting previously committed activities. We may explain this to the user by a warning stating that "After clicking the OK-button, undo will be unavailable for the previously finished steps." Supplemented with a well written help-text, this will probably be understandable.

By the same token, the illogical behaviour of the redoable activity A1 may be resolved by simply committing the top-level xymphony and issuing a warning similar to that of the pivot. This solution, as well as the commitment of a pivot, break atomicity of the process as a whole. Unfortunately, this is an unavoidable problem with today's transaction models.

User Interface Design

One challenge remains. What is the best way to present the undo-function to the user? If a user selects to undo an activity that preceded another activity, he must be informed that the undo will abort both activities. If a user has executed an activity multiple times, how should the system present the choice between each of these instances? And when performing an AND-join, the scope of rollback will be enlarged. How can this be explained to the user in an understandable manner?

It is outside the scope of the thesis to answer these questions here. It will suffice to point them out, and suggest that these issues probably should be considered as early as possible in a prototype. If the user-experience is negative, the advanced functions described in this section may well be unwanted, and implemented functionality remain unused.

Summary of Single-User Mini-Workflows

Xymphonic transactions may be used to implement an undo function for a set of activities that I have chosen to call single-user mini-workflows. In FrameSolutions applications such a mini-workflow is a common building block of workflows. The example task *register case* seems to indicate that the proposed design is suitable for simple task structures.

However, the presence of pivots limits the undo scope and breaks atomicity for the process as a whole. Complex activity structures, such as repeating an activity preceding an AND-split, pose some problems that mainly remain unanswered. Another important issue is to build an understandable user interface for the functionality.

5.3.3 Sequential routing

It should be technically feasible to allow a sequence of activities flowing from one user to the next to be rolled back partially, or as a whole. In discussing single-user sequential routing, I favoured using savepoints rather than subtransactions to achieve a stepwise reverse recoverability. When different users are involved, savepoints are insufficient. A live transaction with savepoints cannot be suspended and taken over by another user. As a consequence, sequential routing between different users requires the design illustrated in figure 5.5 on page 60. It is repeated here for easy reference.

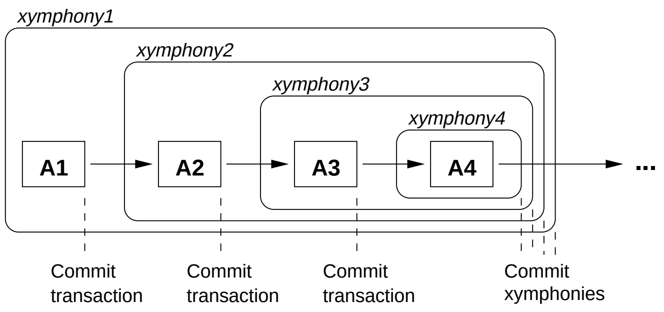


Figure 5.10: Sequential routing between different participants

As shown in the figure, activity A1 cannot be aborted unless A2 is aborted (or not started). Recursively, A2 cannot be aborted unless, A3 is aborted, etc. Aborting the xymphony corresponding to a task takes care of this dependency. For example, aborting *xymphony2*, will roll back the effects of activities A2 through A4.

In LOVISA there is a good probability that an activity will need write-access to data processed by a predecessor. Thus we need to introduce the simple rule that the subtransactions of an activity must commit before establishing a subxymphony for the following activity. This frees the locks of e.g. transaction A1 and by upward inheritance these are retained by the immediate parent. These locks may then be acquired by the activities A2, A3 etc.

For workflow meta models similar to that of FrameSolutions, this design scales to processes as well. Many of the tasks in LOVISA involve only one user, i.e. they contain no AND-splits, and after all the steps have been completed, the next process in the workflow is started by another user. As suggested in the section on mini-workflows, each process should be run in its own task-xymphony. Within each task-xymphony, the individual steps proceed as outlined for mini-workflows. When a task is finished, all subtransactions and subxymphonies are committed to the task-xymphony and a new subxymphony is established for the next task as illustrated in figure 5.10 on the page before.

An Example Case Highlighting some Design Issues

In LOVISA the tasks *register case* and *prepare case* follow each other sequentially. They are illustrated in figure 5.11. Ignoring the optional steps, they involve no AND-splits or pivots.

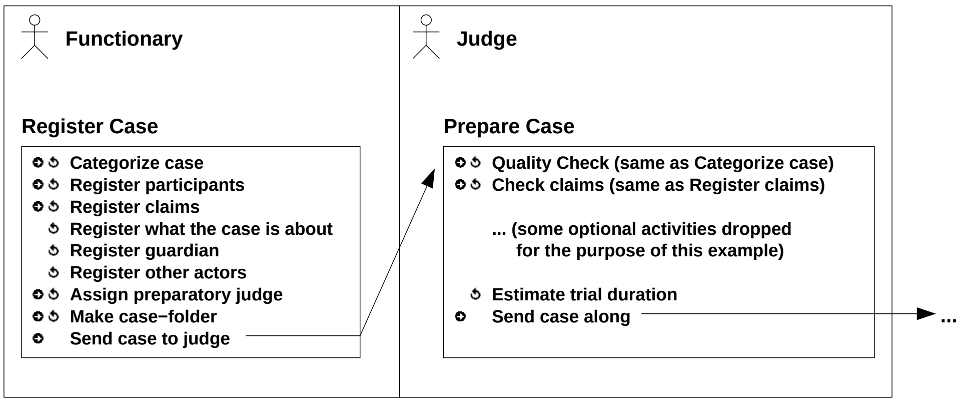


Figure 5.11: Example tasks from LOVISA

Assume that for some case, the functionary has finished the task *register case* and sent it to the judge for preparation. There is clearly a dependency between the tasks, as two of the activities process the same data. In case the judge needs to modify the information registered by the secretary, he must have write-access to the objects processed by the *Quality check* and *Check claims* activities. According to our general design, there would be a xymphony containing the secretary's work, and a subxymphony has been established for the next task. This allows the judge to acquire the necessary write-locks.

Now, assume the functionary decides to undo *register case*, that is, to abort the top-level xymphony. The task *prepare case* would mysteriously disappear from the judge's task list. Any work he had done so far would also be effectively rolled back. We should probably disallow the abortion of other users' tasks to prevent this.

Continuing the example, assume that the judge, after an initial survey of the case, decides that there are omissions that should be corrected by the functionary. He may try to abort his task in order to return the workflow to the functionary. Notice that the last activity completed by the functionary was to *send case to judge*. Consequently, according to our most recent rule, the judge is not allowed to undo this task, as it was not executed by him. Thus, aborting the task would only result in a rollback of *prepare case*, but leave it in the judge's task list.

To work around this problem, we could introduce a property allowing any user to undo a task, and assign it to the activity *send case to judge*. The judge could now be given the option of aborting this last activity executed by the functionary, thereby returning the case. Imagine the functionary's surprise. With no visible explanation, other than possibly a system message saying "rollback complete", the task is back in his task list. He might think he just forgot to *send case to judge*, so he might just do that and we are back to where we started.

To sort out this last problem, we could have the judge fill out a message explaining why he undoes his task. However, I consider this to be a new development in the case, not an abort in the transactional sense. It would be better to use a mechanism already provided in LOVISA. Here, the judge may create a new *register case* task and send it to the functionary with an explanation.

This example illustrates two important points. Having xymphonic transactions makes it technically possible to abort an activity, or a process, and return the workflow to the previous user. But from a user perspective, this would probably seem more like a peculiar error, rather than an elegant functionality.

And secondly, this example illustrates the difficulty in assigning user restrictions on different levels of the subdatabase hierarchy. It seems inappropriate to allow all participants to abort any part of the workflow. But it may be too limited to allow only the abortion of one's own processes. Unfortunately, specifying such rights to a greater detail may require much time and effort.

Drawing Transaction Boundaries

It is not immediately clear where to draw the transactional boundaries between activities. The difficulties elaborated upon in the previous example are partially caused by the assumption that the activity *send case to judge* belong to the task-xymphony of *register case*. The act of sending a case to the next participant will delimit all activities, or processes in a workflow. Should it belong to the predecessor, or to the follower? According to the discussion on access rights, it should belong to both. When analysing recoverability, we'll see that it should in fact be an independent unit between both.

The transition between two users mostly contains system processing. The predecessor initiates the action, and the follower receives a task in his task list. In between, some state in the workflow control data is affected. If this processing is to be undoable as an independent entity, it must have its own subxymphony, or we must be able to compensate it in some other manner.

The transition itself must be independently recoverable. Otherwise, if the transition belongs to the preceding task's xymphony, it must be committed to that xymphony, together with other subtransactions of the task, before establishing the subxymphony for the next task. In that case, the *prepare case* cannot be removed from the judge's task list as part of a rollback, unless the preceding *register case* is also aborted. And if the transition belongs to the following task's subxymphony, it will always be aborted as part of that task's rollback. Thus the judge would be unable to undo his own *prepare case*, without also loosing the task itself (returning the case to the functionary).

For these reasons, a transition from one user to the next should probably be considered an activity in its own right and inserted in the sequence as shown in figure 5.12.

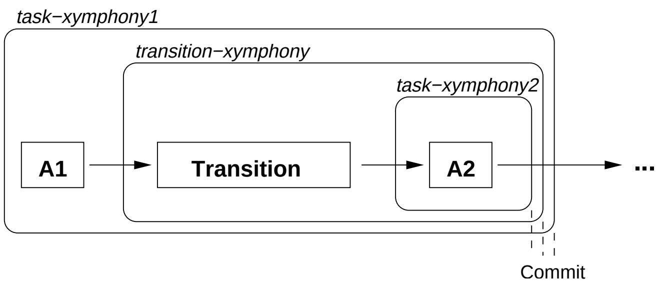


Figure 5.12: A transition may have its own xymphony

The transition-xymphony should by default be undoable by the participants of A1 and A2. Thus the judge would be given the choice between undoing his task only, leaving it in his task list, or he could undo the transition entirely. The functionary would be allowed to undo the sending of the case to the judge, as long as the judge has not yet established the task-xymphony for his task.

The problem with this design is the resource demand when using two subdatabases for each activity instead of one. We could probably use compensating transactions as a means for rolling back the effects of the transition. A compensating transaction would allow the transition to be rolled back even though it was committed to the task's xymphony. It would be the responsibility of the workflow engine to install the compensating transaction for a given transition, and it must also handle the correctness regarding constraints on when a transition can be rolled back (e.g. the task must not have the status *started*). In general compensating transactions are considered to be hard to program, but since the processing between tasks is limited and follows a regular pattern, it should be possible. However, a detailed discussion of this topic is outside the scope of this thesis.

Summary of Sequential Routing

It seems to be technically feasible to process a sequence of activities in a single hierarchy of xymphonies. The design discussed allows stepwise reverse recoverability, including a detailed control over the transition between users. However, designing an understandable user interface is probably a daunting task. The example has highlighted some of the confusion that may rise when a user may abort other users' actions.

5.3.4 Conditional Routing

Conditional routing commences with an OR-split in the workflow definition. At this point a choice between different execution paths are made. The normal interpretation is that as soon as the activity preceding the OR-split is finished, a condition is evaluated to select between which of the following tasks to instantiate and insert in a work list.

The actual execution history will follow a single sequential path. Thus we may utilise the same transaction design as for sequential routing. The evaluation of the condition and subsequent insertion of a task in some user's work list belongs to the transition itself and may be put in a transition-xymphony as suggested in figure 5.12 on the previous page. The following activity is run in a subxymphony.

There is one notable difference between sequential routing and conditional routing. If a task preceding the OR-split is aborted (and all its followers will be automatically aborted as well), then the next time the OR-split condition is evaluated, another path of execution may be selected.

As an example, the subworkflow for managing responses from lay judges (see figure 4.5 on page 41) includes an OR-split. The evaluation of its condition depends on the outcome of the preceding activity *consider application*. If a judge were to undo this task, we must assume that it was to re-evaluate the application. Assume that the first time around, he decided to reject the application. This would create the task *reject application* in the functionary's task list. But on second thoughts, he might decide to reconsider. The abortion of his own task would make the system remove the *reject application* task. Upon reactivation of his task, he could grant the application, thereby inserting the *supplemental selection of lay judges* in the functionary's work list instead.

This is probably an acceptable behaviour, but the developers must be aware of the implications of providing undo-functionality. In cases where an OR-split condition should be evaluated only once, it could be identified as a pivot.

Deferred Choice

van der Aalst (1998, p.20) identifies a special type of conditional routing, the implicit OR-split. The moment of choice is deferred until some event triggers the evaluation of the condition, thus it is also called deferred choice.

van der Aalst, ter Hofstede, Kiepuszewski & Barros (2003, p.34) gives several examples where this pattern would be useful, among them the following; Business trips require approval before being booked. Either the department head approves the trip or both the project manager and the financial manager approve the trip. At the same time both activities are offered to the department head and project manager respectively. The moment one of these activities is selected, the other one disappears.

Few WfMSs support the modelling of deferred choice. However, it can be implemented by inserting both the following activities in appropriate work lists and by cancelling one of them as soon as the other starts (Dumas & ter Hofstede 2001, p.84). A transactional design along these lines is suggested in figure 5.13.

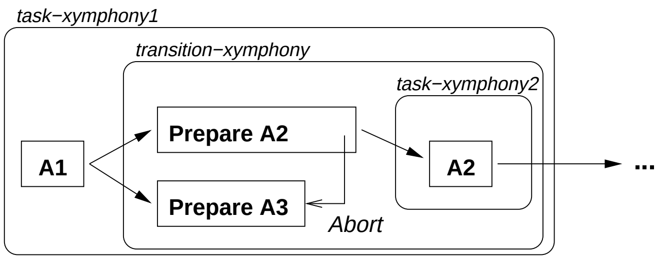


Figure 5.13: One Possible Design for Supporting Deferred Choice

The general idea is that the transition processing inserts two tasks in the respective tasks lists. The transactions that did the actual work are kept active until one of the tasks is executed by a workflow participant. This activation of a task initiates the following actions: The transaction, whose task was selected, aborts the other transaction, thereby removing the now disabled task. It concludes by establishing a new subxymphony for the activity to run in.

General transaction processing systems might not allow one transaction to abort another. However, this can be achieved by signals between the programs that do the work within the above mentioned transactions. Reception of a cancellation signal would make the program issue an abort. To avoid concurrency problems, the sending of messages must be protected by semaphores.

According to my architectural design suggestions, the data that is covered by active transactions should not be externalised, not even by parameterised read operations. If this is followed strictly, the two transactions in figure 5.13 must be converted to xymphonies, and the processing done in short subtransactions. The principal design will still be the same.

Triggers

Deferred choice can be useful for handling triggers in a workflow system based on xymphonic transactions.

LOVISA designs triggers as the starting point of a workflow. In the example given in figure 4.5 on page 41, the subworkflow for managing responses from lay judges is unconnected to the rest of the workflow. However, the two triggering events, expiration of a deadline, or the reception of a response, can only happen as a consequence of activities in the task *select lay judges*. There is also a data dependence between this task and the treatment of responses. *Select lay judges* registers the participation of lay judges, and depending of the responses, this information will be modified.

In order to give the tasks following the triggers the needed write access, they must be run in a subxymphony of *select lay judges*. There will be one triggering event for each lay judge, thus we should probably handle this as a regular AND-split. The workflow engine must be informed of the dependence, so it must be explicitly modelled in the workflow definition.

The modelling itself is relatively straightforward, and an example is shown in figure 5.14 on the following page. There is an AND-split immediately after *select lay judges*. This means that responses from each of the judges will be run in isolation in their respective subxymphonies. Immediately following this there is an implicit OR-split. This means that the processing will wait in the transition-xymphony until one of the triggering events occur. The transition is concluded when one of the events causes either of the following tasks to be created.

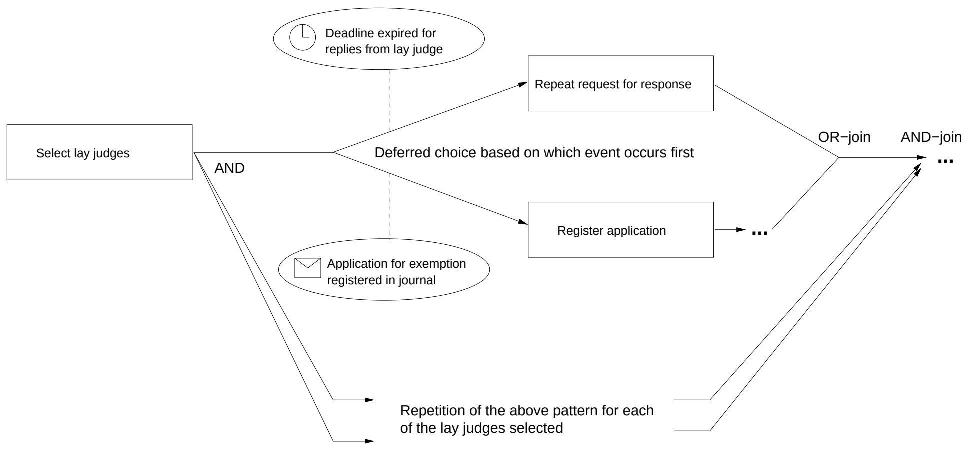

Deadline expired for replies from lay judge
Repeat request for response
Select lay judges
AND
Deferred choice based on which event occurs first
OR−join
AND−join
...
Register application
...
Application for exemption registered in journal
Repetition of the above pattern for each of the lay judges selected

Figure 5.14: Modelling triggering events

5.3.5 Iterative Routing

Iterative routing is a special case of sequential routing. The only difference is that instead of executing different activities, the same activity definition (or set of definitions) is used for making multiple activity instances. As long as the iteration is explicitly designed in the workflow definition, the workflow engine is prepared for the iteration and may interpret it as it would any sequence of activities. This equivalence between sequential and iterative routing is illustrated in figure 5.15.

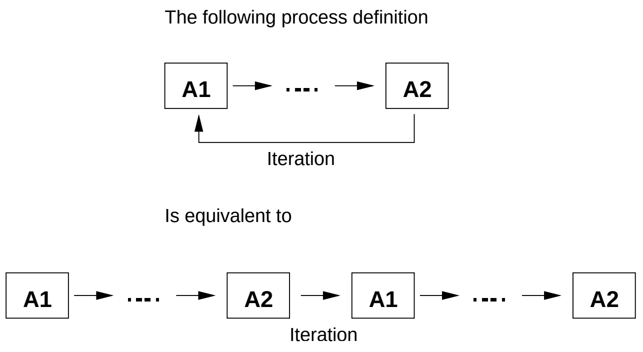


Figure 5.15: Iterative routing is equivalent to sequential routing

Repeated executions of an activity will possibly need access to the same data. Examples from LOVISA include the tasks discussed in the previous section, in which the judge checks the work of the functionary by executing the same activities on the same case. Another example is the collaborative writing of the decision document, which will be passed between the participants in an iteration of the editing activity. Treating an iteration as a sequence will accommodate this by putting the repeated activities in the same subdatabase. This gives the second invocation of an activity access to the same data that was processed by the first.

Redoable activities, however, may present a problem to the correct handling of transactions. As discussed in section 5.3.2, a redoable activity preceding an AND-split may create unforeseen complications to the transaction structure. However, in LOVISA this problem is limited to single-user workflows (i.e. tasks). Iterative routing between different participants must be made explicit in the process definition.

5.3.6 Parallel Routing

Parallel routing implies that two (or more) paths of execution may be executed concurrently for the same case. Parallel routing normally commences with an AND-split and concludes with an AND-join (WfMC 1999*b*, p.29). The AND-split is the routing construct that really strains the limits of adapting long-lasting transactions to workflows. Transactions may be allowed to continue after an AND-split. However, there are circumstances under which it may be more appropriate to commit the transaction before the AND-split.

Assuming that parallel paths of execution need write-access to its data, it is important to determine if they will process disjoint, or conflicting sets of data. If they access different data, we may be able to use the same transaction structure that was proposed for AND-splits in section 5.3.2. If their access patterns conflict, true parallelism is impossible, even if using short transactions. However, we may be able to interleave the execution of short activities within an active xymphony.

The Problem of Predicting Conflicts

When discussing AND-splits in a single user mini-workflow, it was assumed that the activities following an AND-split would process disjoint data. With parallel routing, we may not be able to make such an assumption. The n-way dependence relation will be harder to discover as the workflow becomes large. It may not be apparent in the immediate followers of the AND-split, but an analysis of the transitive closure of the followers, may well reveal that such a dependency is present.

For simple workflows we may still be able to determine that two parallel paths of execution most probably process disjoint sets of data. We may then adopt an optimistic approach, which will result in a delay if a conflict is encountered after all. This is the only design that allows true parallelism.

Transaction Design for Parallel Routing without Conflicts

When there is only a small probability that there will be conflicts between parallel activities, we may adapt the transaction design that was proposed for AND-splits

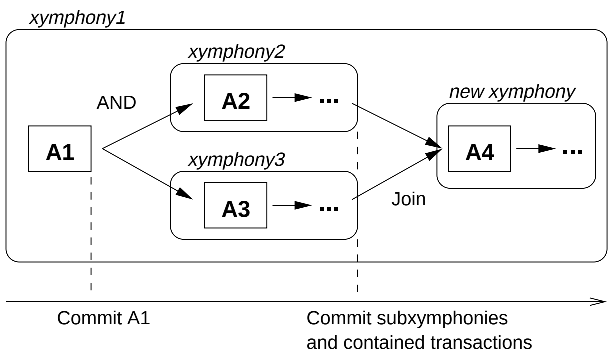


Figure 5.16: The xymphonies resulting from an AND-split

in section 5.3.2. This is shown in figure 5.16.

As is shown in the figure, a subxymphony is established for each of the activities A2 and A3 that will be executed in parallel. The task A1, that precedes the AND-split, must commit its transaction. This allows the parallel activities following to acquire locks (as long as they do not conflict with each other) on the data that was prepared by A1.

Followers of activities A2 and A3 will run in xymphonies 2 and 3 respectively, and may in turn require further nesting according to the rules for sequential routing. There may even be further AND-splits, leading to arbitrary depths of nested databases. This is fine as long as none of the transactions in e.g. xymphony 2 requires locks that conflict with those that are held or retained in xymphony 3.

When the AND-split terminates in an AND-join, the xymphonies 2 and 3, and all contained subtransactions, must commit to the parent. This allows activity A4 and the following to do further processing on the data released by the preceding processes.

Resolving Unpredicted Conflicts

Although the workflow model might give a good indication of which conflicts to expect, they may still occur. We need a way to handle this. Figure 5.17 on the next page shows a situation in which activity A2.2, following somewhere after the AND-split, tries to write an object x, which is locked (held or retained) by xymphony 3.

There are several options for handling this unforeseen conflict.

(1) Wait and/or abort. Upon discovering the conflict, activity A2.2 may wait for the lock to be released by xymphony 3. While waiting it may decide to abort and retry the operation at a later time. Unfortunately the wait may be long. Xymphony 3 could be covering the execution of tasks taking days to finish.

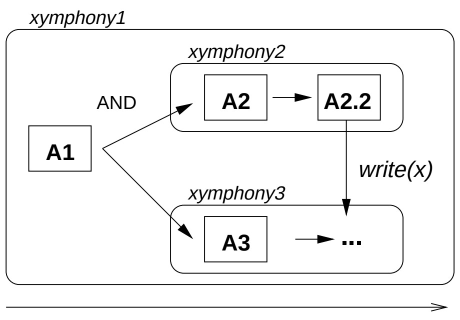


Figure 5.17: An unpredicted conflict between parallel paths of execution

(2) Request or force xymphony 3 to terminate. If xymphony 3 commits as soon as all active tasks finish, the locks will be released. A new xymphony will be established for the remaining activities in the execution path. However, this disrupts the possibility of undoing the activities that were committed, and the object x that was released might again lead to a new conflict between the two paths of execution.

There is also the possibility of aborting xymphony 3, but I don't consider this to be a reasonable solution.

(3) Continue without the requested resource. In some situations, the user may decide that the update he wanted to perform is unnecessary. Especially if the conflict was caused by a pessimistic requesting of write-locks. E.g. in LOVISA, some detailed views will acquire write-locks on the objects before the view is even presented to the user. Cancelling the requested edit-mode may thus resolve some of the situations.

(4) Request export of the resource x. Exporting data is an extension to the original Xymphonic Transaction Model. If xymphony 3 is not actively using x, i.e. the lock is retained by the xymphony itself and not held by an active transaction, it may be exported to activity A2.2. Upon the completion of A2.2, the new value is returned to xymphony 3. This breaks serializability, but it may still be preferable to the options above.

None of these options are ideal ways of handling conflicts. Due to the attempt at making transactions long-lasting, waiting might take a long time, and aborting a xymphony might lead to lots of lost work. In those cases where the user cannot continue without performing the requested update, exporting data between the xymphonies may be a viable solution. However, if conflicts between the parallel paths are common, it will not be a good solution to rely on.

Transaction Design for Parallel Routing with Probable Conflicts

If there are more than a few conflicts to be resolved with exporting, as suggested above, the isolation achieved by using transactions is void. We might as well commit each step as a short transaction in the order that activities are performed. This saves the overhead of maintaining possibly large structures of nested transactions and the delay in waiting for conflicting data to be exported.

We may use the n-way dependency relation in the process definition to tell the workflow engine that a set of parallel activities will need access to the same data. The engine may establish a xymphony for the parallel activities, but all the activities run as flat subtransactions and commit to this xymphony after each step. This will reduce the waiting time when conflicting operations are tried.

Using this approach, only the active transactions may be aborted by normal users. It is not possible to undo previously completed activities individually. In extreme cases, however, the complete xymphony containing the parallel activities may be aborted by an administrator or equivalent super-user.

Example

For an example of parallel routing, we will return to the task *prepare case*. In figure 5.11 on page 65 this task was presented without any of its optional steps. The omitted steps include creation of new tasks for registering interpreters, experts, and defense counsel. In most cases these steps will be performed, thus there is an AND-split from *prepare case*.

Figure 5.18 on the next page is an excerpt of the workflow model for criminal cases. The four tasks to the left register persons that are involved in the case. It is reasonable to assume that the same person will not be registered in different roles for the same case. Thus the tasks are not expected to introduce conflicts between each other.

There could be conflicts between different workflows, if for example the same interpreter was to be assigned to several cases. If this is a rare occurrence, we could handle this by exporting the interpreter's objects from the workflow that initially held the locks. However, the same persons are often involved in different cases, even in different districts running their workflows on different application servers. Vaksvik (2002) suggested that such inter-workflow conflicts could be avoided by a scheme of data partitioning that is described in section 5.5.

Using the transaction design proposed in this chapter, we might get the following structure: The task *prepare case* establishes more than one follower, meaning that all the work executed by its activities must be committed to the task-xymphony. A subxymphony is established for each of its followers.

As noted in chapter 4.3, there are hidden dependencies between the tasks. The task

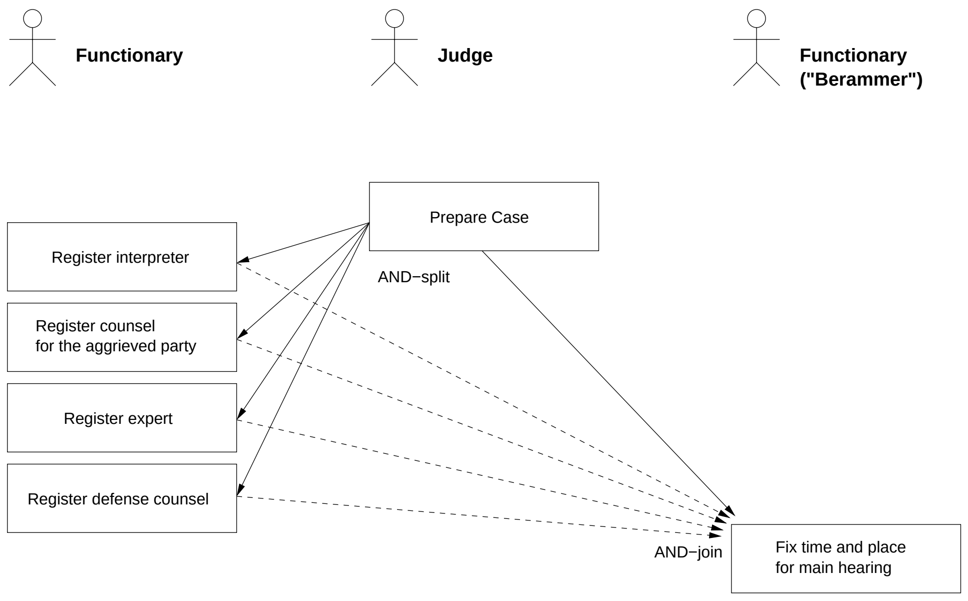


Figure 5.18: Example parallel routing from LOVISA

of fixing time and place for the main hearing includes informing all participants of the hearing. This could be modelled explicitly by creating an AND-join. This is indicated in figure 5.18 by dashed arrows.

When the execution reaches the AND-join, the subxymphonies containing all the five preceding tasks must commit to the parent database. After this point it is no longer possible to undo any of the tasks individually. This constraining behaviour is necessitated by the fact that the objects of the interpreter, expert, and the various counsels are locked in different subxymphonies, and they are needed in the following task when sending letters to inform them of the time and place for the main hearing.

Note that the AND-join described in this example is conditional. The system must make sure that it joins the exact number of parallel paths that were actually instantiated.

Joins and the Scope of Rollback

As the example shows, upon an AND-join, when the subxymphonies of the parallel activities commit, the scope of a possible rollback becomes larger. Referring to figure 5.16 on page 73, if an undo of e.g. xymphony 2 is desired after activity A4 is started, this cannot be done without rolling back the results of xymphony 3 as well.

The historical information that would be required by the system to re-establish these transactions is contained in the log, but would probably require complex and time-consuming analysis to process.

If undo-functionality is the primary motivation for using long-running transactions, we might as well finally commit upon a join. After the commitment of several subxymphonies, the undo-operation becomes an all-or-nothing choice. Something must be terribly wrong for us to undo and possibly loose months of work.

Handling Pivots

Parallel activities executing within a xymphony do not necessarily terminate at the same time. What happens if a task involving a real action asks its xymphony (and all parent xymphonies) to commit while other tasks still have a lot of work to do before they may terminate? The committing task would have to wait for the other tasks to finish, or the other tasks would have to be aborted.

In the context of flexible transactions Leu et al. (1992) suggest that the pivot transaction must wait.

> "A global transaction is *typed* if some of its subtransactions are compensatable and some are not. In a typed transaction, the subtransactions which are compensatable are allowed to commit before the global transaction commits, while the commitment of the non-compensatable subtransactions must wait for a global decision."
>
> (Leu et al. 1992, p.173)

However, there are several situations in which it might be unacceptable to wait for the global decision.

LOVISA heavily depends on the receipt and production of printed documents. Whenever these are sent to external recipients, the event is recorded in the journal and the document must be durably stored in the database. Thus the sending of letters is a typical pivot – Once the mail has left the organisation, it can only be revoked by sending a compensating letter, and the writing of this document cannot be automated by a recovery procedure.

While waiting for the pivot to commit, the user may prepare the documents, and even print them. But he would have to wait for workflow system to signal that the task is successfully committed before delivering them to the postman. Long waiting times would be unacceptable to most users in this situation.

My suggestion is to commit the top-level xymphony before the pivot commences. In this way, the pivot may commit without affecting concurrent processes. The appropriate point at which to commit would be at all AND-splits preceding the pivot activity and whose corresponding AND-joins succeed the pivot.

Summary of Parallel Routing

A xymphonic transaction may continue when the workflow includes parallel execution paths that most probably process disjoint data. Subtransactions preceding the AND-split must commit, and new subxymphonies established for each of the following parallel paths.

It may be difficult to determine that the parallel paths will not process conflicting data. Unforeseen conflicts may be handled in different ways, but if conflicts are probable, all the parallel activities should run as short-duration transactions. These transactions may run in an active xymphony, but the scope of rollback will increase as work progresses.

When the parallel paths merge in an AND-join, their subxymphonies (and subtransactions) must be committed. This also enlarges the scope of a rollback, reducing the usability of long-lasting transactions for the purposes of undo functionality.

Finally, to avoid long waits, the presence of pivots within one of the parallel paths, forces the top-level xymphony to commit, even before the pivot is started.

5.3.7 Summary of Mapping Workflow Models to Transactions

This section has shown that it is possible to support the semantics of the WfMC meta model for workflow definitions to some degree. However, some limitations in the transactional paradigm necessitate the use of shorter transactions due to structural considerations alone. Add to this the limitations on transaction duration discussed in section 5.2 (avoiding lost work and the presence of pivots), it seems that only short and very simple workflows may be encapsulated in one long-lasting transaction. The benefits of using xymphonic transactions will be limited to selected parts of the workflow.

One such selected part is the single-user mini-workflow. Another part of the workflow that may easily benefit from using xymphonic transactions is any subset of tasks that contain no AND-splits. Sequential, conditional and iterative routing may be nested in a transactional structure that supports undo back to a specified point in the execution history. Parallel routing presented some problems for long-lasting transactions. The simple case, in which the parallel paths have little probability of conflicting and contain no pivots, can be accommodated. However, in most other cases, it seems more appropriate to commit before the AND-split.

5.4 The Uses for Parameterised Access Modes

So far, we have primarily discussed the structuring of nested databases in order to accommodate write access between the tasks of a workflow. It may seem that the parameterised read operations of xymphonic transactions are superfluous. It has been suggested that if undo-functionality is the primary motivation for using advanced transaction models, it may be implemented using ordinary nested transactions (Vaksvik 2002, p.95). However, when the duration of transactions increase, there will be a greater probability that other users, apart from the participants themselves, will need to review the information about a case so far.

In fact, upon activating a task, a user may want to open the case and familiarise himself with the developments so far. The task he is about to perform will only need write-access to a fraction of the data that has accumulated, but he may want read access to the rest. In particular, if transactions are long-lasting, he will need read-access to uncommitted data.

The solution is to introduce a default parameter that is used for all the tasks of the workflow. I suggest a parameter indicating that data is the result of a completed activity. Normally in LOVISA, a completed activity is committed to the persistent state. Allowing a default of parameterised access to such information amounts to the same degree of isolation.

The notable difference is that the data is not finally committed, and therefore it may be rolled back. This must be indicated in some way to the user. The application will need a mechanism for distinguishing between committed objects, and objects that were retrieved with parameterised access modes. Annotated nullable logic (see chapter 2.3.4) may be a good choice for this purpose, although a workflow engine implementing its own transaction manager may use custom built functions for determining the transactional properties of an object. However, it is outside the scope of this thesis to elaborate on the implementation details of such a function.

The parameter set may be extended for different purposes. One possibility is to deny access to non-committed values for users that are not directly involved with a case. If only those who are familiar with a case should be exposed to the uncommitted, and therefore uncertain information, we could have the workflow engine assign a parameter for each case and use it only for those clients that have activated a task to operate on the case. The effect is different from that of ordinary access restrictions in that an employee who normally has access rights to the case, might still be prevented from viewing some of the information with the explanation that the case is under development.

A parameterised read is still a dirty read. Some operations will not be acceptable under these circumstances, e.g. updating a bank account. In a workflow that uses a default access parameter, selected activities may be instructed to use an empty set of parameters. The effect is that the activity will run in isolation from other transactions, including its siblings. The data that such an activity has written will be unavailable to others, except those that perform in a subdatabase (i.e. its following activities). The activity can be designated as a pivot, in order to ensure that its followers will only see committed data.

Overriding the access mode in an activity has another possible application. Groups of activities sharing the same set of parameters would be given mutual access to each other's data, whereas any other activities in the workflow would be denied (except in a subdatabase). I have been unable to find useful examples in LOVISA for this application of parameters, but it might come in handy in other settings.

A final way to use parameters, is to have the workflow upgrade its access mode for the remainder of the activities. This behaviour could be achieved by issuing a *set_access_mode* operation to the workflow engine. Such an operation could be user controlled, or it could be automatically executed by an invoked application, possibly corresponding to a modification of the workflow's status. It is tempting to let the parameters indicate the status of the case. However, status is an attribute of the case, taken from the universe of discourse. The status is changed and committed to the database as the case proceeds. A lock-parameter, on the other hand, disappears as soon as the transactions are committed. A parameter to indicate status would only be appropriate as a supplement to the status attribute of the case-object.

5.4.1 Summary of Parameterised Access Modes

Parameterised access modes allow read access to case data that are locked in long-lasting transactions. In general, it may be sufficient to use one simple default parameter for all workflow processes. In some application domains, there will probably be application specific requirements that may benefit from a more advanced use of parameters. However, I have been unable to find any useful examples of this in the LOVISA system.

5.5 Data Partitioning to Avoid Conflicts

Vaksvik suggested that some conflicts could be avoided by partitioning data. Data for which there is high contention and which is changed less often is called *register data* (Vaksvik 2002, p.68). In LOVISA, an example is personal information like birth date, addresses, etc. that will be updated independently of the cases they participate in. This data should be accessed by short transactions to prevent conflicts between workflows, and the updates should be durable, regardless of the outcome of the workflow. Workflow relevant data, on the other hand, must be covered by the transactions discussed above.

Section 5.1 concluded that all data processed in a workflow should be protected by the same spheres of control. The crucial point, which allows us to depart from this position, is that updates to register data are independent of the outcome of the workflow. A failure leading to an abort of the xymphonies would not introduce any inconsistencies between committed register data and aborted workflow relevant data.

The implementation may be tricky. An activity will often process a combination of register and workflow relevant data. For example the registering of participants in a case allows updates to personal information, like name and address (register data), and the object model will be updated with references to these persons and information on which roles they have in the case (workflow relevant data). Upon completion of this activity, the workflow engine would have to sort out the objects that have been defined to be register data and commit them immediately. The rest of the objects, would be committed to the active xymphony, awaiting final commitment until the completion of the top-level xymphony.

In this example, there is a dependency requiring register data to be successfully committed before the workflow relevant data is committed. However, there is not a dependency the other way. Assuming this is the case for all imaginable types of register data, an activity can only succeed if its register data may be committed.

It is possible to prevent a xymphony from committing, if it depends on durable register data. Referential integrity constraints require that a reference to a person can only be committed if the person actually exists in the database. At the time an activity established the reference, it existed. But because access to the person object was covered by a short-duration transaction, it is possible to remove the person from the database before the workflow tries to finally commit. There are several reasons why this person might be removed. A maintenance operation removing old data would not know about the recently made reference to the person. Users accidentally deleting data and sabotage are other possible causes. The result of removing the referenced object would be that the committing workflow would break referential integrity and be forced to abort. Much work could be lost, unless a technique for preventing premature deletes of register data is developed.

5.6 Extensions to the WfMC Process Model

Which extensions to the WfMC Process Model are needed in order to accommodate the transaction design proposed in this chapter? The process definition must contain enough information to enable the workflow engine to automatically establish transactions and subdatabases for the activities to run in. This section outlines some extended attributes that could be added to the workflow definition language proposed by the WfMC (WfMC 1999*a*).

If the workflows are specified in other business procedure definition languages, similar extensions are possible. There are many competitors to the workflow definition language proposed by the WfMC. BPML4WS[2] and BPML[3] are prominent examples, and many others are discussed in (van der Aalst 2003).

[2]http://www.ibm.com/webservices/
[3]http://www.bpmi.org

5.6.1 Process Definition

The first set of attributes applies to the workflow process definition. A process definition, in the terminology of the WfMC, specifies a workflow as a whole, or a subworkflow[4]. A subworkflow allows the process definition to serve as a way of grouping activities. The single-user mini-workflow is one example of a subprocess. Another example is a set of activities that should be processed atomically. The attributes in figure 5.1 control how the workflow engine should handle the process.

Attribute	**Data type**	**Purpose**
use_xymphonies	Boolean	Signals whether long-lasting transactions should be used at all. If true, the workflow will map activities to transactions as suggested in this thesis.
persistent_save-point_interval	time value	If persistent save-points are available, this defines how often a save-point should be set.
commit_frequency	time value	Defines how often the workflow engine should try to commit the top-level xymphony in order to avoid lost work.
default_access_mode	parameter set	The default access mode to be used by all activities of this process.
xymphony_owner	reference to organisation model	Specifies who will be the owner of the xymphony established for this process. If empty, the workflow engine is the owner.
is_single-user	Boolean	All activities in this subprocess are executed by a single user, indicating that the design for single-user mini-workflows is used.

Table 5.1: Attributes relevant to a process definition

[4]In LOVISA a single task is specified in a process definition, an the specification of a workflow is referred to as a *process base*. The WfMC would refer to a LOVISA-task as a subworkflow. However, the difference is in terminology only.

5.6.2 Activity Definitions

The attributes of an activity definition normally specify participant assignment, applications or application statements to run, conditions and restrictions, priority, etc. Routing constructs such as OR- and AND-splits are also defined in the activities (WfMC 1999*a*, p.32). The extensions suggested for transactional control are provided in table 5.2 on the following page.

5.6.3 Workflow Relevant Data

In a standard WfMS, workflow relevant data is “typically used to maintain decision data (used in conditions) or reference data values (parameters) which are passed between activities or subprocesses” (WfMC 1999*a*, p.48). In a system like LOVISA, there is a complete domain model defining all data that is processed by the system. This allows the system full transactional control over most of the data and is probably the best solution for implementing long-lasting transactions. However, the data model of the WfMC’s definition language is not rich enough to express complex object models, or even relational data models. It would therefore be necessary to specify a domain model in a language that is not standardised (so far) for WfMSs.

There are two attributes that give the workflow engine hints on how to manage transactions depending on which data is processed. Regardless of which data definition language that is used, the entities should be extended with the attributes shown in table 5.3 on page 85. For example, if UML is used to design the domain model, they could be incorporated as stereotypes or tagged values.

5.6.4 A Note on Access Modes

All the tables of extended attributes include a *default_access_mode* attribute. If only using a default access mode, as suggested in section 5.4, this attribute should be given the same value in both the process definition (table 5.1) and the definition of workflow relevant data (table 5.3).

If using the design for preventing outsiders access to uncommitted data, the process definition should be assigned a unique parameter for each workflow instance. The default_access_mode could be set to empty for workflow relevant data. The interpretation should be that all operations are unparameterised, unless they are executed in the context of an activity.

As discussed in section 5.4, the workflow enactment service should provide an operation to set the access mode at run-time. However, an activity that overrides the default_access_mode of the process definition, must be allowed to override access modes set at run-time as well. Also note that an empty parameter set is different from a NULL value. An activity running in unconditional isolation would use the

Attribute	**Data type**	**Purpose**
is_pivot	Boolean	This activity must finally commit, as discussed throughout this chapter. In some cases, it is possible to derive the value of this attribute. Interaction with an invoked application, which is considered to be an external system, will automatically be a pivot.
data_dependency	references to other activities	This is the n-way dependency discussed earlier. All activities referencing each other are run as short transactions in the same subxymphony.
default_access_mode	parameter set	Overrides the default access mode for this activity alone.
xymphony_owner	reference to organisation model	Specifies who will be the owner of the xymphony established for this activity. If empty, the workflow engine is the owner.
is_read-only	Boolean	Indicates that a read-only activity may access its data by parameterised read operations. No new subxymphonies are needed for this activity.
default_conflict_resolution	wait for n seconds, commit other, read past, or request export	Applies to activities defining an AND-split. The enumeration of values correspond to the options for resolving conflicts as discussed in section 5.3.6 on page 73.
commit_when_finished	Boolean	For fine-tuning the transactions. Even though an activity is not a pivot, it may be beneficent for the performance to finally commit when the activity completes. It may be particularly useful before an AND-split.

Table 5.2: Attributes relevant to an activity definition

Attribute	Data type	Purpose
is_register_data	Boolean	Indicates partitioning of data as discussed in section 5.5.
default_access_mode	parameter set	Default access mode for operations outside the context of an activity.

Table 5.3: Attributes relevant to the object model

empty set, whereas a NULL, or otherwise undefined value, would indicate that the activity inherits the access mode given in the process definition.

The configuration of access modes at different levels may easily lead to unpredictable conflicts. An activity trying to modify its write parameter, may be prevented from executing, if another transaction has a read-lock on the object. This issue belongs to a discussion of validation and verification of workflow design. Although correct modelling of workflows is an important topic, it is not considered in this thesis.

5.6.5 A Note on Deferred Choice

Unfortunately the deferred choice pattern is not supported by the workflow language specified by the WfMC (van der Aalst 2003, p.77). This may be a problem for handling triggers in xymphonic transactions. The workflow engine must be informed of the data dependency between certain predecessors and the tasks following a deferred choice.

The tables above make no attempt at addressing this issue. It is outside the scope of this thesis to suggest an appropriate extension to support deferred choice.

5.6.6 Summary of Extended Attributes

These extensions to the workflow definition language control the transactions that will be established by the workflow engine. The set of attributes are as simple as possible. As a minimum, a designer must tell the WfMS to use xymphonies, he must identify pivots, and he must provide the n-way dependency relation where appropriate. The rest of the attributes allow fine-tuning of the operation.

Attributes for extending the organisation model have so far been omitted. There should be attributes for defining access privileges to run-time control of workflows. However, there are too many unanswered questions related to this issue. If, and when, these questions are investigated, it should be easy to provide appropriate attributes.

The benefit of extending a standardised workflow definition language, is that any compliant WfMS may enact the workflows. A non-xymphonic workflow engine can just ignore the attributes that it does not understand, and use its own approach to transactions instead. Another benefit is that a graphical tool for defining workflows probably can be modified to model xymphonic workflows quite easily.

5.7 Other Uses for Xymphonic Transactions

So far the approach has been to support the structure and semantics of the workflow definition meta model defined by the WfMC. Xymphonic transactions may also be used to extend the functionality of WfMSs. By using this approach, it is possible to create more complex activities for collaborative efforts, and support ad hoc routing and delegation of tasks. These suggestions may be used even though the system does not implement xymphonic transactions in general.

5.7.1 Documents in Workflow – a Common Special Case

In an office environment, workflow systems will most probably manage documents. A quick survey of WfMS products, reveal that quite a few are specialised in managing document flow. It seems that the producing, editing, reading and distribution of documents is a very common task to be undertaken by a WfMS. As a consequence, benefits of using xymphonic transactions in this area would have a significant impact on the WfMS. The ideas here are based on using *Xymphony for MS Word*, a plug-in allowing collaborative editing of documents to be managed by xymphonic transactions (Anfindsen & Storløpa 2001).

Xymphony for MS Word is activated and controlled from a new menu within Microsoft Word. The owner of the document may partition the document, or the application may convert the structure of chapters, sections, subsections, etc. into a hierarchy of nested transactions. The owner may invite users registered in the system and give them access to work within the context of selected subxymphonies. By using parameterised access modes, the participants may see each other's contribution as work progresses. Upon commitment of a subtransaction, the owner may review the changes before deciding to accept or deny it.

In a WfMS, Xymphony for MS Word might be used as a stand-alone application that is invoked by the workflow client when a user starts an editing activity. The word processor could be given an automatically partitioned document. Lists of users can be extracted from the organisation model describing owner, allowed users, invited users, etc. Invitations to join an editing activity can be sent to the other participants in the form of invitation tasks, automating the process of joining the xymphony. The owner may manually involve other allowed users as appropriate to the circumstances.

In section 4.3, there was an example of collaborative writing in LOVISA. In cases involving more than one professional judge, LOVISA manages the writing of the decision document by routing it between the authors sequentially. Before the writing starts, the judges meet to agree about the content of the document. One judge is responsible for most of the writing, and the other judge makes comments, corrections or additions after receiving the first draft. Still, they have the same authority with regards to the outcome. Using Xymphony for MS Word like outlined above, would allow the writing process to be collaborative. The judges could, for example write different parts of the document simultaneously, and if in doubt, they could submit their part and ask the other to complete it.

A benefit of using Xymphony for MS Word in this way, is that the granularity of the undo mechanism is finer. A workflow routing the document between participants, even if using xymphonic transactions in the WfMS, would only be able to abort at a granularity corresponding to a completed activity. If using Xymphony for MS Word, the scope of undo is document sections, or even finer units if desired. Additionally, for as long as the editing session is active, the undo mechanism built into the word processor may be used as well. If we can save application state, we may even preserve the native undo mechanism between sessions.

The writing of decision documents is the only obvious opportunity for collaborative writing in LOVISA. However, the procedures in an organisation are normally rooted in the traditional way of organising work. With the introduction of an information system allowing parallel processing of cases, the mentality may change. Pipek & Wulf (1999) describes the introduction of a groupware system in a German state administration. They observed that workflows underwent changes even during the use phase, partially stimulated by the technology. They discuss the importance of an integrated view on organisational and technological development. Technology and organisations will both affect each other. It might be possible to identify more processes that could benefit from using concurrent writing of documents, as long as evolution of business processes is supported during the use phase of the system.

5.7.2 Delegating Work

LOVISA supports delegating work by allowing a user to send his task to another. A new task is created in another user's task list, and the original task is deleted. By doing this, however, further control of the task is relinquished.

Using xymphonic transactions, a user may convert his task, or activity into a subdatabase and invite other users to start subtransactions in this context. Such user initiated xymphonies could be exploited in a WfMS to give more control over work that has been delegated. These ideas have been adapted from (Vaksvik 2002, pp.86–88).

The functionality could be designed along the lines of Xymphony for MS Word discussed above. The activity to be delegated must be selected in the work list.

Selecting a task containing many activities, might allow a partitioning of the overall work item into several subxymphonies. The delegation proceeds by inviting users from the organisation model to the various subxymphonies. Such an invitation may create a *delegated task*, which highlights the activities that the invited user is requested to process.

The further execution of the task may proceed in a collaborative fashion. As an example, when executing the task *prepare case* (presented in figure 5.7 on page 61), a functionary might be in doubt as to the correct way of registering claims. He could select the activity *register claims*, invite a more experienced colleague, who in his turn registers the most important claims as an example and submits the work. When the functionary restarts the *register claims* activity, his colleague's contributions are present, and he may complete the activity. The activity may be restarted and work developed iteratively in several more rounds of passing the activity back and forth.

As a final benefit, this way of delegating activities allows the original owner to review the contributions from other parties before accepting them.

Implementing this functionality in a WfMS is a little more complicated than utilising Xymphony for MS Word. The Xymphonic Engine would have to be integrated with the workflow engine. When a user delegates an activity, the workflow engine would have to place the cached copy of the objects to be processed in a xymphony (or subxymphony). The activities of both the original user, and the invited user, would have to be run within the context of subtransactions to this xymphony.

5.8 Comments on Atomicity

It has been an aim to design the transactions so that they may cover as large a part of the workflow as possible. This would give the part atomicity, consistency, isolation (conditional isolation in our case), and upon finally committing, durability. It has been an underlying assumption that a workflow is a logical unit of work that should be executed atomically. This section questions the appropriateness of this assumption.

Indeed, as is shown by the following example from (van den Heuvel & Artyshchev 2002, p.10), there are business procedures that would benefit from being atomic. Figure 5.19 on the following page depicts a business process including the ordering, production, payment and delivery of goods. The concept of spheres of atomicity captures different semantics of atomicity. A sphere must be completed all or nothing, and the execution atomicity sphere covering the whole workflow indicates the desired transactional nature of the entire process.

However, so far researchers in transactional workflows have been unable to provide atomicity to complex workflows. Most systems available today use short transactions. In the case of LOVISA, an application statement is seen as an atomic unit.

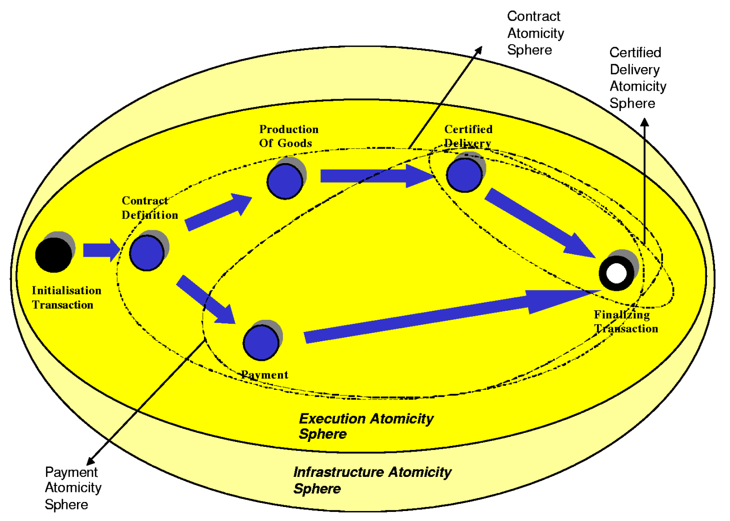

Contract Atomicity Sphere
Certified Delivery Atomicity Sphere
Production Of Goods
Certified Delivery
Contract Definition
Initialisation Transaction
Finalizing Transaction
Payment
Execution Atomicity Sphere
Infrastructure Atomicity Sphere
Payment Atomicity Sphere

Figure 5.19: Spheres of atomicity in an example business transaction (van den Heuvel & Artyshchev 2002, p.10)

From a technical viewpoint this is appropriate. There are other considerations that support such design.

In comparing workflows and transaction monitors, Ader (1997) considers the activity to be the atomic unit. In particular, activities that interact with external transaction oriented systems should conform to the same transaction boundaries as the external system. For the most part, the transactions in the external system are short.

Atomicity, and the complementary property of serializability, might not be appropriate from a business perspective. Eder & Liebhart (1997, p.197) state that “Serializability as a global correctness criterion is not applicable in the workflow domain because business processes themselves are not serial.” Classical transactions are seen as concurrent and completely unrelated units of work, whereas in a workflow there is much cooperation between the different processes. The reduced isolation that is possible with xymphonic transactions accommodates this to a certain degree, but as we have seen, there are problems when the task structures become complicated and the degree of interaction between activities is high (e.g. parallel paths with a high probability of conflicts).

In introducing the concept of *workflow transactions*, Eder & Liebhart (1997, p.199) states that “a workflow transaction is a *sequence of workflow activities* which transfer a business process from one consistent state into the next consistent state. Activities themselves are again workflow transactions.” In this perspective, determining what constitutes the logical unit of operations suitable for one transaction becomes a rather abstract question. An activity, a task or the whole workflow are all candidates. Committing the work done after each activity, as is the rule in LOVISA, corresponds to the view that the business procedure has taken a step forward towards its completion, and the state that it is in, although unfinished, is nevertheless a consistent state.

The attitude taken in this thesis, is that for small workflows it may be appropriate to model the whole workflow as an atomic unit. If there are technical obstructions to reaching this goal, we must look for parts of the workflow that may benefit from being regarded as an atomic unit. In most cases, benefit from a practical perspective (as opposed to a theoretical) will be the driving factor. We may be better served by keeping transactions short if this will lead to better performance, lesser risk of loosing work and simpler technical designs. Balance this with the benefit of providing undo functionality and other benefits of using long-lasting transactions. In a pragmatic approach like this, theoretical properties like atomicity and isolation do not become as important.

5.9 Related Work

5.9.1 Vaksvik 2002

The cand scient thesis by Vaksvik (2002) was a starting point for the work with this thesis. Vaksvik examined the question of integrating xymphonic transactions with FrameSolutions, using Gaius as an example. Gaius is a workflow system for managing cases in the Norwegian Supreme Courts, and in some aspects it is a predecessor to LOVISA. This section will summarise the results of Vaksvik and evaluate how this thesis has developed her ideas further.

Vaksvik (2002, pp.71–73) proposes a *basic model* for the structuring of xymphonies. At the start of a workflow, a top-level xymphony is established to contain the processing of the complete workflow. Each task is run within a subxymphony, and likewise, activities are run within their respective task-xymphonies.

One of the problems identified by Vaksvik is that although the process model fits neatly within the structure of nested databases, work processes do not necessarily follow such hierarchical patterns. Consequently, Vaksvik suggested further analysis of the data processing requirements of a real system.

In the case study of LOVISA, I have focused on the data processed by different tasks, analysing potential conflicts. My approach has been to let dependencies determine the structuring of transactions. The ordering of tasks was identified as the most important indication of dependencies, therefore much space has been devoted to discussing the routing constructs.

Another notable difference is the transaction duration. The basic model covers the whole workflow in one long-lasting transaction. By using shorter transactions, it is possible to support real actions, integration with external systems, and the potential amount of lost work is reduced. The drawback is that the benefits of using xymphonic transactions only apply to parts of a workflow.

A major concern for Vaksvik was to avoid conflicts by carefully structuring and partitioning data. The suggestions on updating register data outside the sphere of control of the workflow-xymphony were taken from (Vaksvik 2002) as discussed in section 5.5. Vaksvik (2002, pp.80–83) includes a detailed discussion of the granularity of locking in the domain model. Central objects, like the case object, are seldom modified, but the references to other objects, e.g. to the documents in a case, are accessed by most tasks. It is important to select the granularity of locking so that an activity editing one document, does not prevent access to, or addition of other documents.

Many of the open questions pointed out in (Vaksvik 2002) remain unanswered. This includes the question of establishing appropriate parameter domains. A default parameter was assumed to be sufficient for most processes, and a more advanced usage of parameters is probably application specific. Finally there is the question of to which degree users should be exposed to the control of xymphonic transactions. Mostly, the workflow engine may manage transactions. However, issues such as user interface design and the assignment of user privileges largely remain unresolved.

5.9.2 Transactional Workflows

In the 1990s researchers were concerned with the lack of transactional support and formally defined failure semantics in current WfMSs. Sheth & Rusinkiewicz (1993) introduced the concept of *transactional workflow* to clearly recognise the relevance of transactions to workflows. Subsequently it has been used by a number of researchers when discussing workflows that define transactional properties for individual tasks or entire workflows (Worah & Sheth 1997, p.9).

The following quote is a typical perspective that motivates the adoption of advanced transaction models for workflow systems.

> "Modern WfMSs have to support complex long-running business processes in a heterogeneous and/or distributed environment. It has been pointed out in [GHS1995][5] that most of these systems lack the ability to ensure correctness and reliability of workflow execution in the presence of failures. Therefore a strong motivation of merging advanced transaction models with workflow models become evident."

(Eder & Liebhart 1997, p.198)

[5]GHS1995 refers to D. Georgakopoulos, M. Hornick, and A. Sheth, 'An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure´, *Distributed and Parallel Databases*, 3(2):119-153, April 1995.

In one approach, tasks are mapped to transactions of an advanced transaction model, and control flow is defined as dependencies between transactional steps (Worah & Sheth 1997, p.9). The ConTract model (Wächter & Reuter 1992) is a good example. A ConTract consists of a set of steps with the ACID properties and an execution plan, called a script. The script specifies execution dependencies between the steps, and may combine multiple steps in atomic units. Each step is required to be compensatable, and the default strategy for dealing with failures is forward recovery. The ConTract model relaxes isolation and atomicity and defines correctness in terms of invariants for each step, leading to *invariant-based serializability*.

However, this focus on the transactional properties of a workflow has its drawbacks. While a "WfMS automates the flow of control and data between activities, and maps activities to users and programs" (Alonso et al. 1996, p.6), transactional workflows address only part of the problem. Alonso et al. (1996) mention the ConTract model as the most comprehensive approach, but still it "does not include users into the system" (*ibid*, p.6).

Furthermore, WfMSs are built for supporting more generic tasks that may be non-transactional in nature, and sometimes even non-computerised (*ibid*, p.6). The systems involved in the processing of a workflow may not provide the services implied by an extended transaction model (Sheth & Rusinkiewicz 1993, p.37). Worah & Sheth (1997, p.5) conclude that "workflow requirements either exceed, or significantly differ from those of ATMs [advanced transaction models] in terms of modelling, coordination and run-time requirements." In their view, the role of advanced transaction models is of a supportive nature (*ibid*, p.31).

These conclusions are valid for the application of xymphonic transactions in workflows as well. The discussion of mapping workflows to transactions has shown that the Xymphonic Transaction Model is not able to conveniently support complex workflows and ad hoc routing (see e.g. the complication caused by the ad hoc AND-join in figure 5.9 on page 63). The transaction model includes users, and may in itself support collaboration. However, it does not provide any way to specify tasks and execution dependencies, thus it addresses only a subset of the requirements for a WfMS. For these reasons, the Xymphonic Transaction Model has been proposed as a supportive extension to WfMSs, adding transactional properties for selected parts of a workflow.

Recent contributions to transaction support for workflows include (Schuldt et al. 2002), from which I have adopted some useful terminology. They provide a framework for specifying and executing processes that are guaranteed to terminate correctly. This is achieved by requiring that, in a process containing a pivot, all activities preceding the pivot must be compensatable, and following the pivot, there must be a least one path of execution that contains only retriable activities. The process may be aborted (by compensation) before the pivot commits. After this, even if all other paths towards termination fail, the path containing only retriable activities may be retried until eventually it succeeds.

The example workflow on page 89 was taken from (van den Heuvel & Artyshchev 2002). Their main discussion centers upon defining different types of atomicity, and how web services may negotiate the degree of atomicity for business-to-business transactions over the Internet. Their approach may provide atomicity to processes that span multiple organisations, however, as is apparent from their plans for further research (*ibid*, p.11), much work remains before this is possible.

Chapter 6

Summary and Conclusions

Transactions have been an important factor in the success of DBMSs for managing data. However, as application requirements have changed, the transactional paradigm has proved too limited for many application domains. In the case of WfMSs, attempts have been made at using advanced transaction models for supporting the structure and inter task dependencies of workflows. However, business processes have richer semantics than what it is possible to express with a transaction model.

Still, WfMSs may benefit from the recovery mechanisms and consistency guarantees provided by transactional systems. This thesis has explored the potential use of the Xymphonic Transaction Model.

The Xymphonic Transaction Model extends the classical transaction model with parameterised access modes and nested databases. Parameterised access modes allow for a controlled access to uncommitted data locked in long-running transactions. Nested databases, or xymphonies as they have been named in the commercial product, are similar to nested transactions, but the sphere of control provides some of the same services to subtransactions as a database would. Xymphonic transactions have all the ACID properties of standard transactions, except for isolation, which is conditional, yielding the acronym ACCID.

6.1 Summary of the Proposed Design

The discussion in chapter 5 addressed the two research questions:

(1) *How must the WfMS be designed to achieve the promised benefits?*

(2) *Which factors constrain the use of xymphonic transactions? Which factors limit the applicability of xymphonic transactions?*

The questions are interrelated, and answers to one, affect the other. The following summarises the design and some of the most important constraints that were found.

The main contribution is a set of rules for transaction handling that may be used by a generic workflow management system to automatically interpret a workflow definition and establish appropriate structures of xymphonies for the processes to run in. However, a precondition is that the WfMS has transactional control over all the data that is processed. This means that the workflow application data must be internal to the WfMS, or that external data sources can be covered by distributed xymphonic transactions. However, many WfMSs integrate external DBMSs, legacy systems, and non-transactional systems. Communication with such systems is restricted to short classical transactions at best. In order to ensure global consistency, interaction with external systems forces the WfMS to commit its internal transactions.

In custom built workflow systems like LOVISA, most of the data processing is internal to the system. This environment is better suited to long-lasting xymphonic transactions. Implementation options for a potential system were briefly discussed, indicating that the proposals in this thesis might be feasible.

The section on mapping from workflows to transactions approached the question of how to structure xymphonic transactions. It is possible to support the semantics of the WfMC meta model for workflow definitions to some degree. Some limitations in the transactional paradigm necessitate the use of shorter transactions due to structural considerations alone. This combines with limitations on transaction duration due to real actions (pivots) and a desire to avoid lost work. It seems that only short and very simple workflows may be encapsulated in one long-lasting transaction. The potential benefits of using xymphonic transactions will be limited to selected parts of the workflow.

One such selected part is the single-user mini-workflow. If a set of activities to be processed by a single user contains no pivots, and the structure of this mini-workflow is relatively simple, xymphonic transactions may provide an undo function to the user. Due to nested databases, such an undo function is more advanced than what would be possible if using only flat transactions with savepoints.

Another part of the workflow that may easily benefit from using xymphonic transactions is any subset of tasks that contains no AND-splits. Sequential routing and iterative routing may be nested in a transactional structure that supports undo back to a specified point in the execution history. The two major problems identified in this context, are assigning appropriate user privileges to the transactional mechanisms, and building an understandable user interface. These problems can be solved by only allowing the user to undo his most recent activity, and restricting access to more complicated control to appropriately trained administrators.

Conditional routing in its basic form is similar to sequential routing. Deferred choice and selection between alternative paths based on triggers require a more careful design. LOVISA has solved this issue by making triggers a starting point in a workflow. However, this simple approach is not possible if the following activities need access to resources locked by the preceding activities. An option discussed is to enable all the paths following an OR-split, but keep them waiting in the transition phase until an event makes a decision possible.

Parallel routing presented some problems for long-lasting transactions. The simple case, in which the parallel paths have little probability of conflicting and contain no pivots, can be accommodated. However, in any other case, it seems more appropriate to commit before the AND-split. A high probability of conflicts and the presence of pivots limit the undo scope and break atomicity for the process as a whole.

Parameterised access modes allow read access to case data that are locked in long-lasting transactions. In general, it may be sufficient to use one simple default parameter for all workflow processes. In some application domains, there will probably be application specific requirements that may benefit from a more advanced use of parameters.

Finally, section 5.6 provided a very simple set of extensions to the workflow definition language for controlling transactions. As a minimum, a designer must tell the WfMS to use xymphonies, he must identify pivots, and he must provide the n-way dependency relation where appropriate. The rest of the attributes allows for fine-tuning of the operation.

6.2 Pros and Cons of Xymphonic transactions in a WfMS

There are both pros and cons for using xymphonic transactions in a WfMS. The following evaluation serves as an answer to the main research question that motivated this thesis: *Which benefits can be had from using xymphonic transactions in a workflow management system (WfMS)?*

Xymphonic transactions may provide the ACCID properties to small workflows, or parts of a larger business procedure. This was one of the aims of researchers in transactional workflows in the 1990s (Rusinkiewicz & Sheth 1995, p.599). The precondition to achieving this, however, is that all the subsystems participating in the atomic unit are protected by the same xymphonic transaction. This condition is satisfied in LOVISA for many single tasks, and for some sets of up to four tasks that do not contain real actions.

We may expect the atomicity semantics of workflows to improve in such a system. However, section 5.8 debated the applicability of enforcing strict atomicity for business procedures, and concluded that practical benefits may be more important than theoretical elegance.

The undo function is one practical benefit of using xymphonic transactions. Based on the built in recovery mechanism of the transaction manager, consistency is automatically ensured. The granularity of the undo-steps is reasonably fine, and in many cases, it allows for individually selecting parallel paths for rollback. However, AND-joins enlarge the scope of rollback, and pivots and other events forcing a commit, prevent further use of undo. Another problem is designing an understandable user interface, and granting appropriate user privileges to transactional control.

Support for long-lasting transactions allows for longer activities than what is possible in LOVISA today. Parameterised access modes ensure that information will be available to other users, even though data is locked. The extra uses for xymphonic transactions further extend the possibilities for complex activities. For example, documents may be edited collaboratively within a single activity, and work may be delegated in a controlled fashion.

A drawback of longer transactions is that more work may be lost in the event of crashes. The availability of persistent savepoints is an important factor in this respect. A parameter for specifying the commit interval was provided to allow organisations to define their own policy on this issue. The risk of lost work may to some extent be compensated for by the undo functionality. Some time is spent by support personnel to recover from user errors in LOVISA. They have repeatedly expressed a desire for a button to undo the error (and all activities depending on it), instead of performing the tedious task of manually correcting it.

The rules provided for automatically mapping workflow tasks to transactions simplifies the workflow specification. Designers do not need an intimate understanding of the Xymphonic Transaction Model, and most of its usage is controlled by a simple set of extended attributes. There is one complicating factor, however. Dependencies between tasks have to be modelled explicitly and more completely than in LOVISA. The designer must identify data dependencies and possibly conflicting parallel paths. AND-splits must be joined in order to inform the workflow engine of the necessary commitment of the parallel paths. And triggers must be connected to the rest of the workflow, usually in the form of an implicit OR-split. LOVISA allows for a more relaxed approach to modelling dependencies.

Xymphonic transactions have limited support for adaptive workflows, and may even hinder the possibility in LOVISA to create ad hoc tasks. An ad hoc task will perform outside the scope of the xymphonies covering the workflow tasks and may experience conflicts with these.

Real time statistics will become more complex when using xymphonic transactions. LOVISA implements statistics by querying the committed state in the DBMS. When using long-lasting transactions, on the other hand, completed tasks may be uncommitted. The statistical function must combine committed and uncommitted data, and there is a question of how to interpret this potentially unreliable information.

These conclusions are based on a comparison of a would-be xymphonic WfMS with LOVISA. LOVISA is representative of WfMSs using classic flat transactions to manage its persistent data. It has several features that are common in WfMSs — it integrates with external systems and implements a workflow meta model similar to the WfMC's standard. On the other hand, LOVISA manages most of its case data internally. This contrasts with WfMSs designed to integrate heterogeneous systems and whose sole purpose is to manage flow control between existing systems, thus not all the conclusions in this section are valid for all types of WfMSs.

6.3 Unresolved Issues and Further Research

Some design issues deserve more attention prior to implementing a WfMS based on the Xymphonic Transaction Model. Following this, there are some suggestions for further research. This distinction is made because the design issues are relatively limited in scope and applicability, although they are important questions to answer when building a xymphonic WfMS.

6.3.1 Unresolved design issues

Performance was assumed to be sufficient for supporting the long lasting transactions proposed in this thesis. Although this may not be true for current systems, increases in hardware performance may alleviate the problem. Furthermore, the orally expressed view of Anfindsen is that a DBMS implementing xymphonic transactions in the core would be far more efficient than a xymphonic transaction manager built as a layer on top of a standard database. Still, final conclusions are difficult to reach without testing the design in a prototype or a simulation.

A detailed discussion of user interface design has been outside the scope of this thesis. It may be beneficent to test this in a prototype and experiment with different variants to be evaluated by users. In particular, the usability of the interface to the undo functions requires further attention.

Another user-related question remains. The suggestions for user privileges to transactional control simply allowed normal users to undo their own tasks. More advanced assignment policies may be useful. However, a comprehensive analysis of this question may require experience with a live system.

A host of lesser details require further attention as well. The architectural designs discussed need further detailing, and other design options may be possible. The design for register data need further development. Locking and the granularity of locking in the domain model has not even been considered in this thesis. Finally, it may be possible to support more complex tasks without implementing a full blown xymphonic WfMS. The suggestions in section 5.7 illustrate a few possibilities, and many similar extensions may be possible.

The final design issue borders on a research question: How can deferred choice be modelled in workflow definition languages that do not support it explicitly? The reason it is listed as a design issue, is that in order to implement a xymphonic WfMS using the standard workflow definition language from the WfMC, it might be sufficient to find a working solution by adding some simple extended attributes to the workflow language.

6.3.2 Further research

The mapping of workflows to transactions (section 5.3) was based on the assumption that dependencies should determine the structuring of transactions. In LOVISA, the ordering of tasks was identified as the most important indication of dependencies. This may, or may not be true for other systems. A case study of several WfMSs would give a better basis for verifying (or falsifying) this assumption.

Adaptive workflows, in particular, present some interesting problems. A workflow definition in which the order of tasks is not given, or only partially defined, does not give the workflow engine enough information about dependencies. Is it possible to determine this information at run-time? Or can the users be given control over the establishing of xymphonies?

Finally, although the extended attributes for controlling transaction mechanisms in a workflow are quite simple, designers may easily make errors. It is possible to define contradictory requirements for different tasks. Developers may fail to identify probable conflicts between parallel paths. The transactions may become too large, locking too much resources and decreasing performance. Or the transactions may become too small to be of any use. Business process management (BPM) may provide a solution to these problems.

van der Aalst, ter Hofstede & Weske (2003) describe BPM as the “next step” of workflow management. BPM includes better support for process design by providing methods and techniques for verification, validation and simulation of the workflows. BPM extends the development process by defining a life-cycle of product design, use, analysis/diagnosis and redesign. The diagnosis phase gather data from the running system an the analysis is input to redesigning existing processes.

While BPM is new and the methods still a research topic, there are many useful techniques for diagnosing workflows. However, adapting these techniques to xymphonic workflows requires further research. A non-exhaustive list of research questions includes: Can formal methods be extended to verify the correctness of the transactional design of a workflow? How can the transactional behaviour of a xymphonic workflow be simulated? Which data should be gathered from the running xymphonic system as input to the diagnosis phase of BPM?

Index

Bibliography

Ader, M. (1997), Workflow engines as transaction monitors, *in* Lawrence (1997), pp. 231–239.

Alonso, G., Agrawal, D., Abbadi, A. E., Kamath, M., Günthör, R. & Mohan, C. (1996), Advanced transaction models in workflow contexts, *in* 'Proc. 12th International Conference on Data Engineering, New Orleans, February 1996.', pp. 574–581.
*A more detailed version of the paper is available as a research report from the IBM Almaden Research Center: http://www.almaden.ibm.com/cs/exotica/exotica_tran_models0795.ps [cited July 2004]

Anfindsen, O. J. (1997), Apotram — an Application-Oriented Transaction Model, PhD thesis, Department of Informatics, University of Oslo.

Anfindsen, O. J. (2002), The power of xymphonic collaboration, White paper, Xymphonic Systems.
*http://www.xymphonic.com/pdf/The power of xymphonic collaboration white paper.pdf [cited October 2002]

Anfindsen, O. J. & Storløpa, R. (2001), Supporting xymphonic transactions on top of oracle, *in* 'SSGRR 2001 Conference Proceedings'.
*http://www.ssgrr.it/en/ssgrr2001/papers/Ole%20Anfindsen.pdf [cited November 2002]

Bernstein, P. A., Hadzilacos, V. & Goodman, N. (1987), *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass.

Bowers, J., Button, G. & Sharrock, W. (1995), Workflow from within and without: Technology and cooperative work on the print industry shopfloor, *in* M. et al. (eds.), ed., 'Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work', Vol. ECSCW'95, Kluwer, pp. 51–66.

Breitbart, Y., Georgakopoulos, D., Rusinkiewicz, M. & Silberschatz, A. (1991), 'On rigorous transaction scheduling', *IEEE Transactions on Software Engineering* **17**(10), 954–960.

Chaffey, P. & Walford, R. (1997), *Norwegian-English law dictionary : criminal law and procedure and other legal terms*, 2nd edition (1st ed. 1992) edn, Universitetsforlaget, Oslo.

Computas (2002), *FrameSolutions Programmer's Guide*, framesolutions/beans version 3.0 edn, Computas AS.

Date, C. J. (2000), *An introduction to database systems*, 7th edn, Addison-Wesley, Reading, Mass.

Davies, C. T. (1978), 'Data processing spheres of control', *IBM-Systems-Journal* **17**(2), 179–198.

Dumas, M. & ter Hofstede, A. H. M. (2001), Uml activity diagrams as a workflow specification language, *in* M. Gogolla & C. Kobryn, eds, 'UML 2001 — The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1–5, 2001, Proceedings', Vol. 2185 of *Lecture Notes in Computer Science*, Springer, pp. 76–90.
*http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2185&spage=76 [cited February 2003]

Eder, J. & Liebhart, W. (1997), Workflow transactions, *in* Lawrence (1997), pp. 196–202.

Elmagarmid, A. K., Leu, Y., Litwin, W. & Rusinkiewicz, M. (1990), A multidatabase transaction model for interbase, *in* 'Proceedings of the 16th International Conference on Very Large Data Bases', Morgan Kaufmann Publishers Inc., pp. 507–518.

Elmasri, R. & Navathe, S. B. (2000), *Fundamentals of database systems*, Addison Wesley, Reading, Mass.

Eswaran, K. P., Gray, J., Lorie, R. A. & Traiger, I. L. (1976), 'The notions of consistency and predicate locks in a database system', *Communications of the ACM* **19**(11), 624–633.

Garcia-Molina, H., Ullman, J. & Widom, J. (2002), *Database Systems: The Complete Book*, Prentice Hall, New Jersey.

Gray, J. & Reuter, A. (1993), *Transaction processing: concepts and techniques*, Morgan Kaufmann Publishers, San Francisco.

Hagen, C. & Alonso, G. (2000), 'Exception handling in workflow management systems', *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* **26**(10), 943–958.

Härder, T. & Reuter, A. (1983), 'Principles of transaction-oriented database recovery', *ACM Computing Surveys* **15**(4), 287–317.

Härder, T. & Rothermel, K. (1993), ‘Concurrency control issues in nested transactions’, *VLDB Journal* **2**(1), 39–74.

ISO (2000), *International Standard ISO/IEC 9075-2:1999, Database Language SQL Part 2: Foundation (SQL/Foundation)*.

Kjølstad, A. G. (2001), Issues concerning parameter sets in apotram, Cand.scient thesis, Department of informatics, University of Oslo.
*http://www.digbib.uio.no/publ/informatikk/2001/1185/kjolstad.pdf [cited September 2002]

Lawrence, P., ed. (1997), *Workflow Handbook 1997, Workflow Management Coalition*, John Wiley and Sons, New York.

Leu, Y., Elmagarmid, A. K. & Boudriga, N. (1992), ‘Specification and execution of transactions for advanced database applications’, *Information Systems* **17**(2), 171–183.

Moss, J. E. B. (1985), *Nested transactions – an approach to reliable distributed computing*, MIT Press, Cambridge, Mass.

Pipek, V. & Wulf, V. (1999), A groupware’s life, *in* S. Bødker, M. Kyng & K. Schmidt, eds, ‘Proceedings of the Sixth European Conference on Computer Supported Cooperative Work’, Kluwer, pp. 199–218.

Reijers, H. A. (2003), *Design and Control of Workflow Processes:Business Process Management for the Service Industry*, Vol. 2617 of *Lecture Notes in Computer Science*, Springer-Verlag Heidelberg.
*http://www.springerlink.com/openurl.asp?genre=issue&issn=0302-9743&volume=2617 [cited February 2004]

Rusinkiewicz, M. & Sheth, A. P. (1995), Specification and execution of transactional workflows, *in* W. Kim, ed., ‘Modern Database Systems: The Object Model, Interoperability, and Beyond’, ACM Press and Addison-Wesley, Reading, Mass., chapter 9, pp. 592–620.

Schuldt, H., Alonso, G., Beeri, C. & Schek, H.-J. (2002), ‘Atomicity and isolation for transactional processes’, *ACM Transactions on Database Systems* **27**(1), 63–116.

Schwenkreis, F. (1996), Workflow for the german federal government – a position paper, *in* A. Sheth, ed., ‘NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions’, National Science Foundation.
*http://lsdis.cs.uga.edu/activities/NSF-workflow/schwenk.html [cited October 2003]

Sheth, A. P. & Rusinkiewicz, M. (1993), ‘On transactional workflows’, *Data Engineering Bulletin* **16**(2), 37–40.
*http://citeseer.nj.nec.com/295779.html [cited June 2003]

Sommerfelt, P. E. T. (2001), Dynamic modification of transaction isolation in the apotram transaction model, Siv.ing thesis, Department of informatics, University of Oslo.
*http://www.digbib.uio.no/publ/informatikk/2001/1341/Sommerfelt2001.pdf [cited September 2002]

Vaksvik, M. S. (2002), Avanserte transaksjonsmekanismer i saksbehandlingssytemer, Cand.scient thesis, Department of informatics, University of Oslo.

van den Heuvel, W.-J. & Artyshchev, S. (2002), ‘Developing a three-dimensional transaction model for supporting atomicity spheres’, Workshop lecture paper presented on Net.ObjectDays 2002 (not reviewed).
*http://citeseer.nj.nec.com/552793.html

van der Aalst, W. (1998), ‘The application of petri nets to workflow management’, *The Journal of Circuits, Systems and Computers* **8**(1), 21–66.
*http://tmitwww.tm.tue.nl/staff/wvdaalst/Publications/p53.pdf [cited February 2004]

van der Aalst, W. M. P. (2003), ‘Don’t go with the flow: Web services composition standards exposed’, *IEEE Intelligent Systems* **18**(1), 72–76.
*http://tmitwww.tm.tue.nl/research/patterns/download/ieeewebflow.pdf [cited April 2004]

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B. & Barros, A. (2003), ‘Workflow patterns’, *Distributed and Parallel Databases* **14**(3), 5–51.
*http://tmitwww.tm.tue.nl/research/patterns/documentation.htm [cited February 2004]

van der Aalst, W., ter Hofstede, A. & Weske, M. (2003), Business process management: A survey, *in* ‘Conference on Business Process Management: On the Application of Formal Methods to Process-Aware Information Systems’, Vol. 2678 of *Lecture Notes in Computer Science*, Springer, pp. 1–12.

van Leeuwen, F. (1997), Learning from experience in workflow projects, *in* Lawrence (1997), pp. 185–193.

Wächter, H. & Reuter, A. (1992), The contract model, *in* A. K. Elmagarmid, ed., ‘Database Transaction Models for Advanced Applications’, Morgan Kaufmann Publishers, chapter 7, pp. 219–263.

WfMC (1995), The workflow reference model, Technical Report WFMC-TC-1003, Issue 1.1, Workflow Management Coalition, Hampshire, UK.
*http://www.wfmc.org/standards/docs/tc003v11.pdf [cited March 2004]

WfMC (1999*a*), Interface 1: Process definition interchange process model, Technical Report WfMC TC-1016-P, Version 1.1, Workflow Management Coalition.
*http://www.wfmc.org/standards/docs/TC-1016-P_v11_IF1_Process_definition_Interchange.pdf [cited May 2003]

WfMC (1999*b*), Terminology and glossary, Technical Report WFMC-TC-1011, Issue 3.0, Workflow Management Coalition, Hampshire, UK.
*http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf [cited May 2003]

Worah, D. & Sheth, A. P. (1997), Transactions in transactional workflows, *in* S. Jajodia & L. Kerschberg, eds, 'Advanced Transaction Models and Architectures', Kluwer Academic Publishers, pp. 3–34.
*http://lsdis.cs.uga.edu/lib/download/WS97.pdf [cited March 2004]

Yearly Statistics for the Courts of Justice (2002), Norwegian Courts Administration.
*http://www.domstol.no/Domstolene/internet/showLinks.asp?archive=1002200 [cited January 2003]