UNIVERSITY OF OSLO
Department of Informatics

Combining the Xymphonic transaction model with versioning features

Stian Bakken

Cand Scient Thesis

5th May 2002

Preface

This thesis is a mandatory part of the five year Cand. Scient. degree at the University of Oslo, Norway. The workload of writing the thesis is designated to be one year's worth of full-time studies.

My interest and fascination for database systems was seriously ignited when as an undergraduate student, I attended a course named "Data oriented systems development and relational databases". The course was taught by Professor Ragnar Normann, who is well known for his inspirational lectures. After this experience, I continued to pursue my interest in database systems.

This interest made it natural for me to look for probable thesis topics in the same field, and when I became aware of an available thesis description on transaction management I jumped on it.

I would like to thank my advisor, Ole Jørgen Anfindsen, an expert on transaction management. Most of all for his kind, constructive criticism, but also for debating my ideas with me, for offering me a part-time thesis related job, and finally for giving me the degree of freedom under which I like to work.

Also, my friends and family deserve my gratitude, particularly my parents for their neverending support.

Finally, I would like to thank my wife, Silje, for her love, patience and support. She has also earned kudos for coping with my non-standard working hours in the final stages of this writing.

Oslo, May 5, 2002
Stian Bakken

Abstract

It is a well known fact in the database community that the ACID properties of transactions are too restrictive for long lasting collaboration efforts. Anfindsen (1997) shows that the only ACID property we really want to compromise is that of isolation. He presents a solution to the problem of long lasting transactions, namely the Xymphonic transaction model.

It is argued in this thesis that there are many good reasons for being able to create versions in general, and more specifically in connection with Xymphonic transactions. The main result of this thesis is a solution of how the Xymphonic model can be combined with features from the field of versioning. It is shown through a prototype implementation that the proposed solution is fully functional. The solution, however, does not claim to be either optimal nor complete, and a chapter is devoted to outlining some interesting areas of further work.

Contents

Chapter 1

Introduction

1.1 Problem definition

Main problem

Is it possible to combine Xymphonic transactions with principles from the field of version management in a way that is beneficial for one or both fields?

Adherent problems

In order to give a satisfying answer to the main problem, some other problems need to be solved along the way. At least the following three questions should be considered:

1. How, if at all, can versions be generated in a controlled fashion in a tree of transactions and Xymphonies?
2. How, if at all, can configuration control be incorporated into this scheme?
3. Is it possible to create a model which is also feasible to implement?

1.2 Research methods employed

- **Litterature studies**
  In order to draw knowledge from existing versioning models and techniques, an extensive study of research litterature has been carried out.
- **Theoretical experimentation**
  Unfortunately, the best idea is seldom also the first. In the pursuit

of combining Xymphonic transactions with versioning, different approaches have been tested.

- **Practical programming and experimentation**
  A prototype implementation was also carried out. This resulted in some slight modifications of the model.

1.3 Thesis structure

- Chapter 2 constitutes the prerequisite reading in transaction models for the rest of the thesis. It does not contain any new information or discoveries, but rather gives an overview of relevant problems, models and techniques in transaction handling.
- Chapter 3 gives an overview of the Xymphonic model, the advanced transaction model which serves as the basis for the work presented in the thesis.
- Chapter 4 is chapter 2's version management counterpart. It gives a quick overview of the most common terms, techniques and motivations for versioning.
- Chapter 5 gives a description of a conceptual model which integrates version and configuration management with Xymphonic transactions.
- Chapter 6 suggests some possible lines of further research stemming from the discoveries presented in chapter 5.
- Chapter 7 summarizes the main contributions of the thesis. It also ties the thesis together in a conclusion with respect to the propounded problems.
- Appendix A lays out a description of a concrete, practical implementation of the model. It also describes some relevant concerns experienced during the course of the implementation and an argumentation for the different implementation specific choices.
- Appendix B presents notation for describing allowable nesting, as well as instances of versions and relations between versions, in a more formal manner.

Chapter 2

Database management concepts

2.1 Database properties

In the beginning paragraphs of "An introduction to database systems" by Date (1995), a database (DB) is metaphorically defined as a "kind of electronic filing cabinet". A DB can range from a collection of files, to arbitrarily complex structures. Informally, a DB is to be regarded as a collection of persistent data in some order. By persistent is implied that the data will survive the termination of instances of applications connected to the database.

In addition, we would like to access the DB in a safe and controlled manner. This includes concurrency control between independent actors in the system, and the existence of mechanisms to recover data in the case of some failure. Our interaction with the DB is controlled by the database management system (DBMS) which is described below.

2.2 Database management system properties

Most textbooks on database fundamentals have a good description of DBMSs. Established sources are for example (Date 1995, Elmasri & Navathe 1989).

The DBMS should provide certain services regarding management of the DB. This management can be decomposed into the following items:

- **Access layer**
  The DBMS should provide application programmers and users with

an interface layer through which they can access the data in the DB.

- **Security**
  It is desirable to have functionality, such as management of users, authentication and authorization, and cryptography features. This should be provided by the DBMS.

- **Concurrency control**
  In a multi user environment where user interactions with the DB are running in parallell or pseudo-parallell, there is bound to be entanglements of access patterns. The DBMS should make sure that these patterns don't result in inconsistencies or other undesirable effects on the DB.

- **Recovery**
  The DBMS should provide us with a safety-net in case of failure. It is, however, impossible for the DBMS to guard against all possible kinds of failure. Date (1995, page 380) divides database failures into two categories:

  - **System failures**
    System failures affect the users accessing the DB and their work which has not yet been commited. But it does not cause any damage to the DB itself. Examples of system failures can be power failure, operating system halt/crash, etc.
  - **Media failures**
    This class of failures differ from the system type in that occurences of it causes physical damage to the DB. At a minimum, it causes damage to the work in progress of users which need access to the affected portion. Examples encompass hard drive failure and catastrophes of nature.

  Media failures can not be avoided with the help of a DBMS; they must be remedied by restoring a backup of the affected region of the DB. This can, for instance, be either a tape backup or a RAID-based solution. The DBMS should, however, be able to help in case of a system failure. Due to this, system failures are sometimes referred to as *soft* failures while media failures are considered *hard* failures.

Of the mentioned items, the latter two, namely concurrency control and recovery, are handled by the use of database transactions and will be explained later in this chapter.

2.2.1 Database systems

There are different types of database systems (DBS), e.g. distributed and centralized. In a distributed setting, the users may need to connect to many different DBSs. Distributed database systems (DDBS) raise many complicated issues regarding synchronization, which centralized database systems (CDBS) do not. For the sake of simplicity (and to keep the workload at a reasonable level), the focus of this thesis will be on CDBS scenarios. Material on distributed database transactional concepts is found, for example, in (Kim 1995, chapter 28).

2.2.2 Motivation for using database systems

To begin with, using electronic storage methods for data, as opposed to traditional paper document handling, gives the following benefits (Date 1995):

- **Compactness**
  No need to store heaps of paper in filing cabinets as they can be crammed in volumes into the DB.
- **Speed**
  Both storage, retrieval and esp. queries can be done a lot faster with electronic storage.
- **Automation**
  Tedious, manual tasks can be programmed to be performed automatically.
- **Currency**
  Accurate access to the most recent data.

A natural next step is to compare the use of a DBS to conventional file-system methods. The following arguments, in favor of database systems, are taken from (Kjølstad 2001[1], Date 1995):

- The features of a DBMS are powerful arguments:
  - **Access layer**
    The access layer provides us with data abstraction. That is, the storage of the data is de-coupled from the actual applications. This makes it easy for different types of applications to access the same data, and a change in an application does not implicate a change in the conceptual way the data is stored and accessed.

[1] Which in turn references (Silberschatz et al 1997, Elmasri & Navathe 1994).

  - **Security**
    To provide authorization/authentication and privilege levels is a powerful feature.
  - **Concurrency**
    A DBS lets users connect and perform parallel operations in an interleaved, controlled manner. This eliminates data inconsistencies between applications and increases performance.
  - **Recovery**
    If we are interrupted in the middle of a task due to some soft failure, we can be sure that the the DBS has mechanisms to recover to a previously recorded consistent state.

- Features more closely related to the storage of the data:
  - **Reduced redundancy**
    Compared to applications which each have its own private files, a DBS will yield less redundant storage.
  - **Enforcement of standards**
    Standardized data storage can be enforced when there is central control of the DBS. This is particularly important for interchange of data between systems. Date (1995) also mentions data naming and documentation standards as very desirable properties for data sharing and understandability.
  - **Data integrity**
    Integrity can be ensured by applying the proper constraints to the schema of the DB. For example, it is undesirable that an employee is marked to be working in a department which does not exist[2]. Note that integrity only becomes important when we have a DB with redundancy.

2.2.3 Evolution

In the earlier days of DBMSs, when the inverted list, network and hierachical datamodels were the competing standards, applications and DBMSs were tightly coupled in the sense that for an application to interact with the DB, it would have to use procedural access directly on the DB strucure. On the network datamodel, (Bachman 1965) is a classical source. This low level interface model made the task of creating an application and maintaining a database controllable, but still tedious

[2]For a relational database system, this would typically be a case where we would like a foreign key constraint to ensure that only departments from a list of valid departments could be added to other relations.

and time consuming.

With the publication of Codd's article (1970) on structuring the data based on mathematical relations, DBSMs gained some very powerful features. The article suggested a tabular ordering of the data. Database management systems based on these ideas are known as relational (RDBMS). Compared to the earlier efforts, RDBMSs provide a higher level of abstraction and add mathematical rigor to the underlying datamodel. Important features of RDBMSs include the following:

1. The data were now stored with a structure based on mathematical relations which made it natural to view the data in a tabular ordering. For implementations of RDBMSs, pointers and lower level mechanisms were of course still in use, but this was now transparent to users and application programmers.

2. It was possible to attach relational constraints to ensure data integrity.

3. The data could be queried in a *nested* fashion, because each operation would return a table which again could be queried in the same expression. The most widely used query language is the structured query language (SQL). Other examples are query language (QUEL) and the less formal query by example (QBE).

The next leap forward came with the merging of object oriented (OO) and DBS concepts. The OO paradigm was invented and developed in the 1960's with the programming language Simula67, see (Dahl et al 1970, Birtwistle et al 1973). Later, with successor languages such as C++, Java, Smalltalk, Eiffel, C# and many more, OO programming proved to be a big success. Based on the good experiences with the OO methodology and the desire to reap the advantages listed below, object oriented DBMS's (OODBMS) were constructed.

Advantages of OODMSs include:

1. They retain many of the desirable features of RDBMSs.

2. Customizable datatypes are easily constructed.

3. They make it possible to connect datatypes in the DB with each other in more complex ways.

4. Natural integration with programming languages.

Examples of OODBMSs are O2, Objectivity, Poet FastObjects, ObjectStore, Versant and numerous others. There have also been attempts to

provide RDBMS with OO mechanisms and this class of DBSs is known as object relational DBMSs (ORDBMS).

In the future, the demands on database mechanisms, scalability and algorithms are expected to change dramatically. This is based on the rapid growth of data, internetworking, distributed computing, and the expectation of the emergence of new technologies (Silberschatz et al 1991, Eisenberg & Melton 1999).

If we take a look at the kind of tasks the DBMSs were designed for in the beginning days, we see that the companies that realized the powers of using a DBMS were typically of the financial/commercial type[3] who needed to store data about their customers, their purchases, purchase history, and internal data concerning funds, disposals, and so on.

The data processing tasks these companies relied on were commonly very fast and operated on relatively small fields of data. Consequently, if two employees were to access the same field in the DB, the waiting time the second one had to endure would be negligible.

As the DBMSs evolved and got more sophisticated, so did the application areas. Not all these new application domains were well catered for by the traditional concurrency mechanisms, as will be explained next.

2.3 The concept of transactions

2.3.1 A definition of transaction

Basically, a transaction is a sequence of operations we can decompose to read and write operations. These operations are to be performed on one or more databases by a DBMS. However, to ensure consistency and recoverability of our data, we need a handling strategy for transactions. A classical strategy is the ACID approach.

2.3.2 The ACID properties

The ACID acronym stands for Atomicity, Consistency, Isolation and Durability. It originates from an article by Härder and Reuter (1983). In their discussion of what a transaction is, they state:

> The concept of a transaction ...requires that all of its actions be executed indivisibly: Either all actions are properly

[3] Examples of which are banking, airline and insurance.

> reflected in the database or nothing has happened. ... To achieve this kind of indivisibility, a transaction must have four properties:
>
> *Atomicity.* It must be the all-or-nothing type desribed above, and the user must, whatever happens, know which state he or she is in.
>
> *Consistency.* A transaction reaching its normal end (EOT, end of transaction), thereby commiting its results, preserves the concistency of the database. In other words, each successful transaction by definition commits only legal results. This condition is neccessary for the fourth property, durability.
>
> *Isolation.* Events within a transaction must be hidden from other transactions running concurrently. If this were not the case, a transaction could not be reset to its beginning for the reasons sketched above. The techniques that achieve isolation are known as synchronization, and since Gray et al. [1976] there have been numerous contributions to this topic of database research [Kohler 1981].
>
> *Durability.* Once a transaction has been completed and has commited its results to the database, the system must guarantee that these results survive any subsequent malfunction. Since there is no sphere of control constituting a set of transactions, the database management system (DBMS) has no control beyond transaction boundaries. Therefore the user must have a guarantee that the things the system says have happened have actually happened. Since, by definition, each transaction is correct, the effects of an inevitable incorrect transaction (i.e. the transaction containing faulty data) can only be removed by countertransactions.
>
> These four properties, atomicity, consitency, isolation, and durability (ACID), describe the major highlights of the transaction paradigm, which has influenced many aspects of development in database systems. We therefore consider the question of whether the transaction is supported by a particular system to be the ACID test of the system's quality.

These are definitely well thought over properties, and they are desirable in most database application areas. However, for certain application areas, isolation becomes too strict a demand (Anfindsen 1997, 6-16).

2.3.3 Long lasting transactions

Especially in the 1990's, the need for long lasting transactions (LLT) evolved due to new kinds of applications utilizing DBMSs (Kaiser 1995, Nodine & Zdonik 1992).

Computer aided design (CAD), computer aided manufacturing (CAM), computer aided software engineering (CASE), software configuration management (SCM), document collaboration and different kinds of interactive services are often mentioned as good examples of such applications.

With the isolation level provided by the ACID compliant transaction models used by the earlier DBMSs, the duration transactions spent waiting to access data resources was intolerable. Without modifactions made, DBMSs would be useless for these new application areas. There has been considerable work done in the area of LLTs, for example:

Approaches described by Kaiser (1995):

- Versions and checkout/checkin, as in e.g. SCCS (Rochkind 1975) and RCS (Tichy 1985).
- Configurations of versions, as in e.g. domain relative addressing (Walpole et al 1988).
- Semantic coordination, as in e.g. Smile (Kaiser & Feiler 1987) and Infuse (Kaiser et al 1989).
- Optimistic coordination, as in e.g. NSE (Honda 1988).
- Walter's control spheres (1984).
- Notification, as in e.g. the Gordion database system (Ege & Ellis 1987).
- Split-join, dynamic restructuring of transactions by Kaiser and Pu (1992).
- Transaction groups for collaboration, by Hornick and Zdonik (1987).

The Xymphonic model for collaborative transactions will be explained more in-depth in the next chapter. In essence, it caters to LLTs by:

- Customizing the isolation levels.
- Making dynamic sharing of resources possible by exposing them to other users through xymphonies.

2.4 Transaction histories

This section explains some characteristics of transactions and different classes of transaction histories, also known as transaction schedules. For a more detailed treatment, see (Anfindsen 1997). Bernstein et al (1987) have a formal and thorough treatment of transactions. A less technical approach is found in Date (1995) esp. pages 374 to 410.

The most central concept in the field of transaction theory is that of serializability (SR). A serial history is merely a history where *no* interleaving of operations from different transactions are allowed. In order for a transaction history to be serializable, it must be equivalent to a serial history by some criteria.

As suggested, there are different kinds of serializability, most notably view (VSR) and conflict (CSR), where VSR has only theoretical interest, since determination involves solving an NP-complete problem. In practice, all systems are CSR based.

So, what do we do to make our transaction history CSR, and what does it mean that two histories are conflict equivalent? Anfindsen (1997) states that two histories are conflict equivalent if:

1. They contain the same transactions and the same operations.
2. Conflicting operations of non-aborted transactions are ordered the same way in both histories.

For a more precise definition of what it means for two operations to be in conflict, see (Buchmann et al 1992).

In order for a transaction history to be CSR, we need DBMS mechanisms which can guarantee this. Examples of which are:

1. Serialization graph (SG) testing
2. Timestamp ordering
3. Locking

2.4.1 The serialization graph

The serialization graph of a transaction history is constructed by making a node for each commited transaction. For each conflicting operation issued there should be an edge going from the node issuing the first of the conflicting operations to the latter.

If we have a cycle, it follows that conflicting transactions can't be CSR ordered. Consider the simplest example with two transactions where number two has issued a conflicting operation causing an edge from one to two. This is ok. But if afterwards, transaction one were to issue a conflicting operation with transaction two, this would imply that transaction two must precede transaction one, while on the other hand the first edge implies that transaction one must precede transaction two in a serial history. This is absurd and the history is not CSR.

The observation that a transaction history can only be serializable iff the SG is acyclic is formalized in what is known as the fundamental theorem of serializability (Bernstein et al. 1987, page 33).

2.4.2 Timestamp ordering

Timestamp ordering (TO) works by assigning timestamps to each transaction and also assigning to each data item the timestamp of the transaction which last read or wrote them. By comparing these timestamps (Anfindsen 1997, 9), an acyclic SG can be guaranteed.

The TO rules can be summed up as follows: Firstly, a transaction can only read an item whose write timestamp is older than that of the transaction. Secondly, a transaction can not write a data item unless both its read and write timestamps are older than that of the transaction. For a cycle in the SG to occur, a transaction would need to have a timestamp less than its own timestamp. This is impossible, and a serializable history is guaranteed (Bernstein et al 1987, 114).

2.4.3 Locking

Locking is by far the most commonly used means of concurrency control, and the common algorithm is referred to as two phase locking (2PL).

The concept of locking is that in order to perform an operation on a data item, the transaction must be holding an appropriate lock on the particular item. Different lockmodes are explained in section 3.3.

2PL divides the lifespan of a transaction into two separable phases. In the first phase, the transaction acquires locks. At this stage, actions are allowed to be performed as long as the transaction holds the relevant locks, but no locks are allowed to be released.

When the transaction releases its first lock, it enters the second phase. It can still perform actions under the same restrictions as before, but under no circumstances can it request a new lock, as this would compromise the guarantee of a CSR history.

2.4.4 Classical types of failure

When many users access the database simultaneously, different kinds of anomalies may occur if there is no control on the execution:

- **The lost update problem**
  We have two transactions (trA and trB), four points in time (t1 . . . t4 | t1 < . . . < t4), and three values for a data item in the DB (v1 . . . v3). At t1, trA reads v1 from the DB (with an intent to later update it with a new value). At t2, trB also reads v1 (before trA has updated it, but also with an intent to update its value). At t3, trA updates v1 to v2. The problem occurs at t4 when trB updates the same entry with its new value, v3, overwriting v2, written by trA (which assumes the update went ok, and that the value is v2). The update is lost, hence the name. Note that this situation is impossible if we can guarantee a CSR schedule.

- **The uncommited dependency problem / dirty read**
  For the actions taken by a transaction to be reflected permanently in the database (durability), we must guarantee atomicity. This is done by commiting the changes made by the transaction at the end of its lifespan. Let us also here consider two intertwining transactions, trA and trB. At some point in time before its commit, trA writes some value v1 into the DB. Then (also before trA is commited) trB reads that value, assuming its validity. A problem arises if trA needs to be aborted. In that case trB has made itself dependent on an uncommited value, which in this case also is invalid. We can say that it has performed a dirty read. If we have a series of dirty reads and the first transaction aborts, we get cascading aborts, which is considered a very undesirable outcome. If transactions are not allowed to read lock values written by uncommited transactions, we would avoid cascading aborts, and the transaction history would belong to the class ACA (Anfindsen 1997, page 19)

- **The inconsistent analysis problem**
  This problem typically arises when some transaction, trA, is performing an aggregate function on some collection of values from the DB. Simultaneously some other transaction, trB, performs a transfer of a value by decrementing one item and incrementing

the other, in this situation from an attribute already read by trA to an attribute not yet read by trA. This makes the outcome of the analysis performed by trA inconsistent with the contents of the database.

- **The unrepeatable read**
  This occurs when a transaction reads an item twice and the item is changed by another transaction between the reads, i.e. the reading transaction is unable to repeat its first read.

- **Phantom rows**
  Suppose a transaction first retrieves a set of rows all satisfying a predicate. Then another transaction inserts a new row satisfying that same predicate. If the first transaction now runs the query again, a new row appears which did not exist the first time. This new row is called a phantom.

These problems are explained in more detail, with illustrations, in (Date 1995, 393-399).

2.5 Chapter summary and conclusions

This chapter has contained a summary of some basic elements, techniques and problems from the field of transaction management. It has provided a suitable backdrop for the next chapter, which elaborates on the Xymphonic transaction model.

Chapter 3

The Xymphonic model

The purpose of this chapter is to give an overview of the features, concepts and ideas of the Xymphonic transaction model[1]. It is an advanced model tailored to meet the demands of collaborative, long lasting activities. These demands include a desire to cut down the duration potentially spent waiting to access a resource with a traditional model, and also the ability to communicate the quality (e.g. maturity) of data items.

Carefully note that there are many types of collaborative activities, and that the Xymphonic model is designed for those we call data-centric. That is, they evolve around the molding and processing of data.

Other approaches, apart from the Xymphonic, have been proposed to solve the dilemma of long lasting collaborative transactions. In his chapter on related work, Anfindsen (1997, 97-102) gives a short description of 17 other models whereof these are a subset: Sagas (Garcia-Molina & Salem 1987), Acticities/Transaction Model (Dayal et al 1991), ConTract (Wächter & Reuter 1992) and Split transactions (Kaiser & Pu 1992).

This chapter will not give any detailed description of other models than the Xymphonic, neither will it rate Xymphonic transactions against any of the approaches mentioned above. This is based on the fact that the thesis will not be expanding Xymphonic transaction with features from other transaction models, but rather augment it with versioning features.

Structurally, this chapter is designed to first clarify some terms of the Xymphonic model and give a motivation for its features. The chapter then proceeds to describe the major contributions of the model. Where

[1]The words Xymphony^{tm} and Xymphonic^{tm} are registered trademarks of Xymphonic Systems AS.

appropriate, related and incorporated concepts will be introduced. A short summary is given at the end of the chapter.

3.1 A clarification of terms

In order to avoid any possible later misunderstandings, this section is intended to sort out some possible sources of confusion with respect to Apotram and Xymphonic.

Apotram is an abbreviation for application oriented transaction model. It is the topic of Anfindsen (1997), Phd thesis and defines a new transaction model with special support for long lasting, collaborative data-centric transactions. The "application oriented" part of the name is due to its ability to change isolation levels between transactions depending on the specific need of the application. Also, it has support for situations where more than one actor wants to edit a resource (although not at once) through the use of nested transactions and databases.

The name **Xymphonic** was later invented in connection with the commercialization of Apotram technology. In this thesis, Xymphonic transactions will replace the older term Apotram and Apotram transactions. Also, to make Apotram technology more appealing and intuitive for the end-user, nested database was replaced by Xymphony but they are conceptually identical. The Xymphonic parlance and concepts are described in a white paper by Anfindsen (2002), and in (Andfindsen & Storløpa 2001).

Throughout this thesis, Xymphonic and Xymphony will replace the corresponding terms in older Apotram papers. This is based on the desire to stay current with development and papers from Xymphonic Systems AS.

3.2 Motivation for using the Xymphonic model

The first goal of this section is to show the major shortcomings of traditional ACID transactions when it comes to collaborative LLTs. These shortcomings will then be related to a corresponding solution in the Xymphonic model.

The second goal is to show shortcomings of standard version management models and how they are alleviated by Xymphonic transactions. As a rule of thumb, ACID has too strict isolation while version manage-

ment systems tend to be too permissive.

Anfindsen (1997, 6-7) points out that the only property of ACID one should want to compromise is isolation". In order to clearly see why, desirable properties of a collaborative system will be compared to those of ACID.

We would like the following:

1. To look at uncommited results in a safe, customizable manner. This means that we do not desire this reading to result in any inconsistencies and that it would also be beneficial to be able to give directions as to what state the data should be in before we read it.

2. To write to resources held in a conflicting mode by other users in a safe, controlled manner. Just as the preceeding item, we don't want the operation to lead to inconsistencies.

With ACID transactions, those wishes are impossible to grant due to the impenetrable isolation between transactions. In section 3.3, the most common lockmodes and their compatibilities are described. Because the Xymphonic model uses locking for concurrency control, locking and lock incompatibilities are the natural way of showing how Xymphonic transactions differ from ACID.

If we look at item 1 listed above, we must realize that this is impossible to accomplish with ACID due to the fact that reading and writing are conflicting operations that they must be completely isolated from each other, end of story. With Xymphonic, this conflict is solved by making the conflict conditional based on a choice of lock parameter set. This solution is descibed in more detail in section 3.4.

Just as reading and writing conflict, nobody should be surprised that write operations conflict with each other, and that they thus are impermissible when the ACID properties are the correctness criteria employed. Alas, ACID does give room for leniency in the situation in item 2 listed above. Xymphonic transactions solve this type of conflict with what is known as nested conflict serializability. This solution is described in section 3.5.

In order to consider version management models and their solutions according to the two desired features, we must unfortunately generalize a bit. Note, however, that efforts have been made as to construct a unified framework for version configuration management (Westfechtel

et al 2001, Belkhatir & Conradi 1996).

The overall impression is that of the actors in the general myriad of different version management tools, most would merely allow a dirty read, using the eqivalent compatibilities of what is known as a browse lock, see section 3.3 to provide reading of write locked items. Two drawbacks are that we do not get any metadata about the data element, and that we mostly do not get as current data as we would like.

As for the controlled write cooperation, this would normally be handled by creating parallel versions known as variants, or branches, of the data. Variants are useful in many situations, but for the situation in item 2 listed above, the following are some of the drawbacks:

- The variants would later have to be merged, and there is no general algorithm to perform this without user interaction.
- Merging is tedious and also error prone in that it is possible to overwrite the work of others.
- The parallel writers have no clue as to what the other one is writing. They may not even know that they are working in parallel and performing potentially conflicting edits.

3.3 Lock types and their reciprocal compatibilities

Read and write locks will be extended later in this chapter with parameters, and in chapter 5 two new types of locks will be introduced. As an introduction, the traditional lock matrix as presented in (Anfindsen 1997, 20) and its lock types will be explained. Intent locks are used when we would like to apply locks at different granularities from a single transaction. The general algorithm involves placing a lock on a higher level resource with a recorded intent to maybe later lock lower level resources in a more exclusive mode.

The traditional lock matrix

The matrix, shown in figure 3.1, is symmetric, indicating that the lockmodes are symmetric, i.e. it does not matter which lock is set first when deciding if modes conflict. However, schemes with asymmetric locking are conceivable (Anfindsen 1997, 105-106).

The following list gives a short summary and explanation of the different lockmodes in the matrix:

	B	IR	R	U	IW	RIW	W	X
B	×	×	×	×	×	×	×	
IR	×	×	×	×	×	×		
R	×	×	×	×				
U	×	×	×					
IW	×	×			×			
RIW	×	×						
W	×							
X								

Figure 3.1: The traditional lock matrix with intent locks.

- **B - Browse**
  Browse locks are intended to be used in situations when we would like to allow dirty reads.
- **IR - Intention to read**
  This lockmode is used with different granularities and indicates an intent to read lock a resource on a lower level.
- **R - Read**
  Read locks are used when we want to read the contents of a resource and avoid dirty reads.
- **U - Upgrade**
  The purpose of this lockmode is increased concurrency. It is motivated by the fact that it is unfair to block readers in cases where we would like to read contents and *maybe* later perform some write operations . Then other readers can read in the meantime until we decide to upgrade our read lock to write.
- **IW - Intention to write**
  This lockmode is used to signal an intent to write lock a resource on a lower level.
- **RIW - Read with an intent to write**
  The RIW lock is used to place a read lock on a higher level resource with the intent to later write lock lower level data.
- **W - Write**
  This is the standard lockmode used for writing to a data object.
- **X - Exclusive**
  Exclusive locks are used when absolutely no other access can be allowed to a resource. This can for example be when we want to

drop a table in an RDBMS or delete a collection of objects from an OODBMS.

The intent locks are described in more detail by Gray and Reuter (1993, 406-409). In short, they are used to increase concurrency and to avoid to occurence of phantom rows.

3.4 Conditional Conflict Serializability

Conditional conflict serializability (CCSR) is a generalization of CSR as defined in chapter 2, and it is defined in (Anfindsen 1997, 29-43). This generalization is done by applying parameter sets to the different lock-modes and deciding for which combinations of locks and parameter sets there should be a conflict. Clearly, this relaxes the isolation property of ACID.

The general rule is that for two arbitrary parameter sets, A and B, within an arbitrary parameter domain, D, we say that the read lock, R, with parameter set A, denoted $\mathrm{R}(A)$, conditionally conflicts with the write lock, W, with parameter set B, denoted $\mathrm{W}(B)$, unless $B \subseteq A$.

In order to see how this is a generalization of CSR, consider the situation where A is the empty set, $\varnothing$, and B is $*$, an arbitrary superset of D. Consequently the locks $\mathrm{R}(\varnothing)$ and $\mathrm{W}(*)$ will always conflict as we would expect for write and read locks in a CSR history. That is, CSR is retained as a special case.

Two transaction histories are defined to be conditional conflict equivalent by adding the word conditional to the definition for conflict equivalence:

- They contain the same transactions and the same operations.
- Conditionally conflicting operations of non-aborted transactions are ordered the same way on both histories.

In order to extend the definition for two histories to be CCSR, the property conditional is added. Anfindsen (1997) gives the following definition:

> A history is defined as *conditional conflict serializable* (CCSR) iff it is conditional conflict equivalent to a serial history.

More specifically, to determine whether two operations are conditionally conflicting or not, consult the lock matrix on page 32 in (Anfindsen 1997). Among the advantages of CCSR over standard CSR, the following may be noted:

- Read/write conflicts are made conditional thereby providing for a controllable level of isolation.
- CSR is still retained as a special case.
- Users have the opportunity to communicate the quality, e.g. maturity, of their work.
- It makes new types of queries based on access parameters possible (Anfindsen 2002)

As a conclusion of this section, a small example will be given. In the example, the following parameter sets are used: A = { incomplete }, B = { complete }, C = { incomplete, complete }. Consider Alice and Bob who are working on a project together. Bob has a W(A) lock on resource R. If Alice wants to look at Bob's work she will have to accept the fact that R is in an incomplete state and hence choose to ignore it or otherwise lock it with an arbitrary set equal to or superior of A. If Alice is only willing to accept R(B) the lock can not be granted. R(C) and R(A) are examples of conditionally compatible locks, and can thus be granted.

3.5 Nested Conflict Serializability

In order to provide a structured solution to conflicts between writers, nested conflict serializability was invented (Anfindsen 1997). It relies on the concepts of sphere of control (SOC) (Davies 1978) and nested transactions (Davies 1973, Moss 1981). The solution employs nesting of sets of data with special properties, these are called Xymphonies.

3.5.1 Spheres of control

The following explanation of the concept of SOC is largely based on (Andindsen 1997, 45) and (Gray & Reuter, 174-180). Central to SOC is the concept of an abstract data type (ADT). Ford & Topp (1996) characterize ADTs by the following properties:

- It exports a type.
- It exports a set of operations. This set is called the interface.
- Operations of the interface are the only access mechanisms to the type's data structure.
- Axioms and preconditions define the application domain of the type.

Spheres of control relate ADTs to transactional concepts. Gray & Reuter (1993) write on page 174 that:

> Any system that wants to employ the idea of spheres of control must be structured into a hiearchy of abstract datatypes.

It is understood that each invocation on these ADTs is an atomic operation from the perspective of the caller. Also, it should be possible to dynamically expand SOCs to hold results from invocations, pending commitment of data.

SOCs shall not be amplified to any greater extent here, except the important observation that SOCs are a powerful vessel of logical data division, and at the same time keeping track of dependencies between shared data and messages. This makes the SOC a powerful and highly customizable concept, but as Gray & Reuter (1993) point out on page 180, it has never been fully formalized. However, in connection with Xymphonic transactions it will provide us with a logical view of the resources on which we have read locks (RSOC), and the resources we have write locked (WSOC)[2].

Note that the transaction in itself may be viewed as a SOC, as can a query. The following section will contain some more concrete examples of SOCs.

3.5.2 Nested transactions

The concept of nested transactions date back to Davies (1973) and the first comprehensive design was the work of Reed (1978). However, Reed used timestamps for synchronization and it is was not before the Phd thesis of Moss (1981) and the use of locks that nested transactions received any greater attention.

Gray & Reuter paraphrase[3] Moss' (1981) definition of nested transactions as:

1. A nested transaction is a tree of transactions, the subtrees of which are either nested or flat transactions.
2. Transactions at the leaf level are flat transactions. The distance from the root to the leaves can be different for different parts of the tree.

[2] RSOC and WSOC are examples of data SOCs.

[3] The expression flat transaction is used to denote the standard form of transactions as described in section 2.3.

3. The transaction at the root of the tree is called the *top-level transaction*; the others are called *subtransactions*. A transaction's predecessor in the tree is called a parent; a subtransaction at the next lower level is also called a *child*.
4. A subtransaction can either commit or roll back; its commit will not take effect, though, unless the parent transaction commits. By induction, therefore, any subtransaction can finally commit only if the root transaction commits.
5. The rollback of a transaction anywhere in the tree causes all its subtransactions to roll back. This, taken with the previous point, is the reason why sub-transactions have only A, C, and *I*, but not *D*.

In this thesis, root transactions will be named top level transactions (TLT). It should also be noted that nesting of transactions come in two different types. Namely open nesting and closed nesting. Open nesting is characterized by subtransactions executing and commiting independently of their parent transactions. This may require compensating transactions in case of rollback. With closed nesting, subtransactions must begin after their parent transaction and finish before them. Also, the commitment of subtransactions will be dependent on the outcome of the parent. Unless otherwise stated, closed nesting is assumed.

An example of a nested transaction

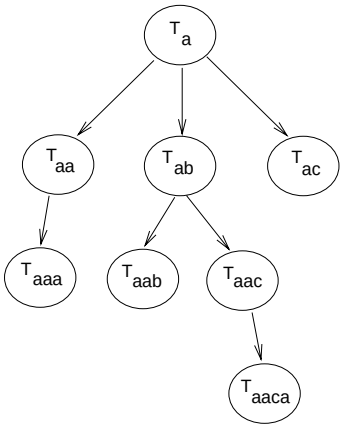

Figure 3.2: A tree with nested transactions.

The root transaction of this simple tree is T_a. The transactions T_{aa}, T_{ab} and T_{ac} are the children of T_a and therefore subtransactions. Carefully note that all transactions in the tree except the root, T_a, are subtransactions. T_{aaa}, T_{aab} and T_{aac} are an example of siblings because they belong to the same level in the tree. The transactions T_{aaa}, T_{aab}, T_{aaca} and T_{ac} are the leaf transactions. The transaction T_{aa} is the parent transaction of T_{aaa}. In the model by Moss (1981), only leaf transactions may perform work, the others function as control structures.

Advantages of nested transactions over flat transactions

This section paraphrases reasons given in (Anfindsen 1997, 25-27).

- **Increased performance**
  If we take advantage of nested transactions, tasks within the transaction may be executed in parallell if they are assigned different transactions in the tree. This is called intratransaction parallellism. Attempts have been made to provide flat transactions with parallellism, but this approach has flaws as described by Anfindsen (1994). It should also be pointed out that using nested transactions does not provide help with parallellism among TLTs, called intertransaction parallellism.

- **Distribution**
  Each transaction in the tree provides us with a suitable unit for distribution.

- **Access control**
  It is conceivable that a parent transaction can control its child transactions in various ways. This can, for example, be with respect to resource access privileges.

- **Encapsulation**
  A parent transaction does not have to know anything about the interior or further spawning of subtransactions of a subtransaction, only that it gets the desired result, so it is fair to say that subtransactions are encapsulated.

- **Recovery control**
  If a part of a flat transaction fails, we can always recover to a previous savepoint. This affects the entire transaction of course. However, with nested transactions this is not the case. If a subtransaction experiences failure, it can do a rollback like a flat transaction but this has no effect on any other part of the nested transaction. Also, if it has to abort, it does not neccessarily need to have a

catastrophic effect. Due to this, transactions in the nested structure are commonly said to have firewalls.

- **Security**
  A transaction in the nesting provides a suitable unit for authorization control.

A practical example of using nested transactions

This is a small example on ordering spare parts for a broken car. A TLT called $T_{orderparts}$ is started, and also subtransactions called $T_{suspension}$, T_{motor}, $T_{antenna}$. Later it turns out that the transaction in charge of ordering new suspensions, $T_{suspension}$, was aborted. Fortunately, this does not have any bearing on the other transactions in the tree as far as the model of nested transactions is concerned, and they can continue their work without any notice. The TLT may wish to start a new $T_{suspension}$, and wait until all the subtransactions have commited before itself commits.

It should be noted that it would be extremely desirable for the transactions in this nesting to be able to share their data and communicate their progress. This is provided for by using Xymphonies, which is explained in the next section.

3.5.3 Xymphonies

The treatment of nested transactions should leave something to be desired. For many application domains we must expect (sub)transactions to be interested in sharing their data, and even collaborating on editing efforts across traditional transaction boundaries. The answer to this with regard to the Xymphonic model is given with the concept of Xymphonies.

Recall section 3.5.1 where SOCs were defined. It would certainly be advantageous if a transaction was able to convert a set of write locked resources, i.e. a WSOC, to another type of data SOC with similar semantics as the global database. This is exactly what a Xymphony is designed to be.

A Xymphony has the following characteristics:

- It must be owned by a transaction.
- In order to be created, a WSOC of a transaction must be converted to a Xymphony.

- Users may be invited to participate in the Xymphony and acquire locks and thus create RSOCs and WSOCs as they would in normal interactions with the global DB.
- The invited users may convert their local WSOCs to new Xymphonies within the Xymphony. This yields a recursive nesting of Xymphonies.
- A Xymphony will be locked with a special type of lock, called a DB lock. This lock has the same conflict properties as a write lock.
- When a participant in a Xymphony commits a WSOC, the work is commited to the Xymphony, not the global DB. It can therefore be perused by the Xymphony owner, who has the power to ratify or deny, i.e. commit or abort, the work.
- A Xymphony can only be eliminated when no locks are held within it (Kjølstad 2001, 40).

The correctness criterion NCSR

Nested conflict serializability (NCSR) is the second correctness criterion in the Xymphonic model. For NCSR, CSR should be enforced as correctness criterion in a nested manner. Anfindsen (1997) defines NCSR on page 49 by making the following requirements:

- Xymphonies can be nested to arbitrary depths.
- Transaction histories in Xymphonies are CCSR.
- Transactions in Xymphonies commit to the Xymphony owner.

This definition and explanation will suffice for the use of NCSR in this thesis. It should also be clear from the definition that CSR is a special case of NCSR.

An example of collaboration through NCSR Xymphonies

Let us consider a small scenario with three human actors, Alice, Bob and Stan. Figure 3.3 illustrates a common database scenario where no Xymphonies are in use.

However, for some good reason Alice would like to delegate some of her work to Bob. In order to accomplish this, she converts part of her WSOC to a Xymphony and invites Bob, who correspondingly starts off a subtransaction inside the Xymphony and creates a WSOC by write locking some resources, see figure 3.4. Stan was not invited to participate in

the Xymphony and continues his work as before.

When Bob finishes his part of the collaboration, he commits the subtransaction $T_{Bob_{aa}}$ and its WSOC is commited to $T_{Alice_a - Xym_a}$. Alice decides to ratify Bob's work and his changes will therefore be reflected in

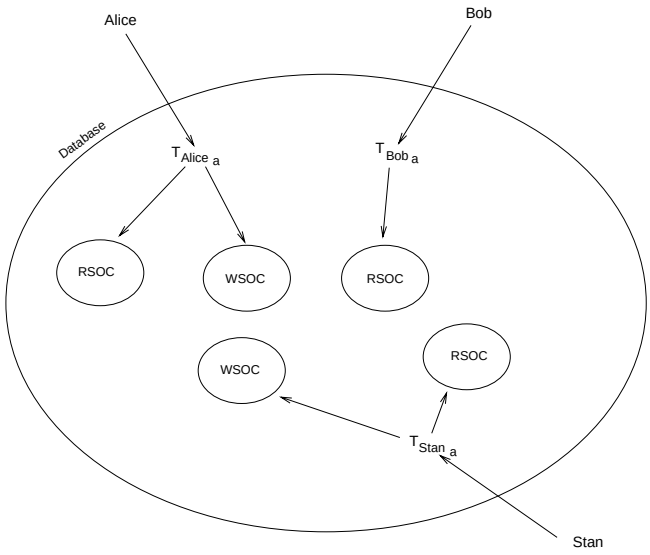

Figure 3.3: Common database scenario, no Xymphonies.

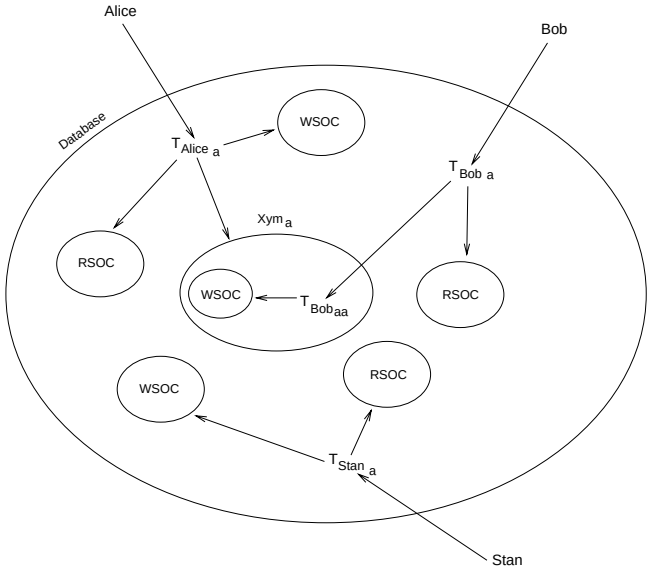

Figure 3.4: Xymphonic cooperation scenario, NCSR.

the contents of the Xymphony. When Stan decides to commit his transaction, the resulting action is that the contents of his WSOC are reflected in the global DB.

3.6 Nested Conditional Conflict Serializability

If we combine the two correctness criterions CCSR and NCSR we get nested conditional conflict serializability (NCCSR), which is the third and final correctness criterion of the Xymphonic model. This combination implies that histories of Xymphonies should use CCSR instead of CSR as correctness criterion. This seems to be only a minor adjustment, but in effect it results in a powerful solution of both read/write, write/read conflicts due to CCSR and also write/write conflicts due to NCSR. Anfindsen (1997) defines NCCSR on page 50 by making the following requirements:

- Xymphonies can be nested to arbitrary depths.
- Transaction histories in Xymphonies are CCSR.
- Transactions in Xymphonies commit to the Xymphony owner.

If those requirements are met, we have a NCCSR transaction history. It should be clear from the definition that CSR, CCSR and NCSR are special cases of NCCSR.

An example of collaboration through NCCSR Xymphonies

This example will expand the example given in section 3.5.3, where we considered collaboration through the use of NCSR histories to also allow NCCSR histories. In figure 3.4, only Bob was invited to $T_{Alice_a - Xym_a}$. In this case Stan will also be invited as portrayed in figure 3.5.

Let us imagine that Stan and Bob were invited in order to collaborate on some design where Alice is the project manager. This can, for example, be the design of the suspension of an automobile design. Alice has grouped the pertinent data in her Xymphony, and Bob and Stan are both actively participating in the Xymphony. It is not hard to imagine that after a while, Bob would like to see how things are going with Stan's design and vice versa. By allowing Xymphony histories to be CCSR, they can use parametrized lock modes and browse each others work in a controlled fashion.

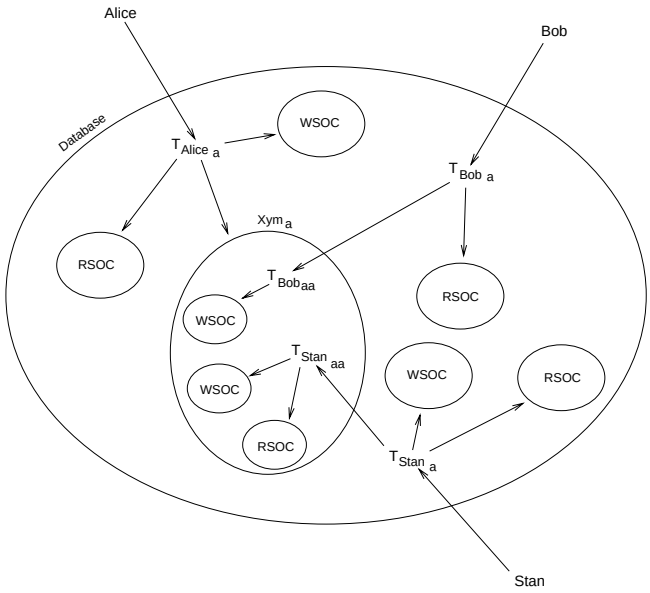

Figure 3.5: Xymphonic cooperation scenario, NCCSR.

Maybe at some later point, Bob realizes that he does not have the required skills to complete parts of his work. Then, by allowing NCSR histories, he can just convert the particular elements to a new, nested Xymphony and invite the users with suitable skills and knowledge to complete the part.

3.7 Chapter summary and conclusions

In this chapter the basic underpinnings, motives and foundation for the Xymphonic transaction model have been explained. The purpose of the chapter has been to show why we need Xymphonic collaboration, what it takes to extend traditional transactions to allow for Xymphonic histories and finally how this has been accomplished.

The three correctness criteria of the Xymphonic model have all been introduced. Conditional conflict serializability, CCSR, is a generalization of CSR which makes read/write and write/read conflicts conditional, based on user preference. Nested conflict serializability, NCSR, introduces the notion of Xymphonies as a solution for write/write conflicts. Finally, nested conditional conflict serializability, NCCSR, is explained as a sensible combination of CCSR and NCSR which remedies all occurences of write and read conflicts.

Chapter 4

Version management concepts

When I started working on my thesis, I was under the impression that versioning was a quite small and well understood area in computer science. I soon realized that I had been gravely mistaken. The management of versions is a large and diversified field, and it has implications on all development projects of any degree of seriousness which involves the use of computers. Also, I soon found that considerable research still needs to be performed in this area, and that it was not so well understood as I had initially thought. Fortunately for me and my work, there exist articles summarizing and classifying work done in the field of versioning models, in particular (Conradi & Westfechtel 1998, Dart 1991, Feiler 1991).

During this chapter in particular, and the rest of the thesis in general, I will use the following lingual conventions when talking about version-related subjects: Firstly, object in this context means basically every kind of entity we can work on and save in computer storage. Secondly development will refer to general modifications pertaining to these objects.

This chapter is structured to introduce the definitions early, and then proceed to the discussion of the different approaches to versioning. This is to make the terminology clear first, because the later sections depend on a good understanding of the basics.

4.1 Defining versioning

It is not easy to give a very precise definition of versioning, since it is quite dependent on the versioning model, and the context it is used

in. This compressed definition however, is based on the one given by Munch (1993):

> Versioning is the managing (storage and retrieval) of versions of objects, as opposed to just managing the objects themselves. By object, anything versionable is implied.

Note that versioning can also be used as a term for the user's act of creating multiple versions of the same object.

Most people will have an intuitive understanding of what is meant by "version", so it will not be attempted to formalize an absolute definition. Informally, a version can't exist by itself, but must be understood as being a version *of* something. What I mean is that based on collected version information, we are able to construct a concrete instance of some object. This construction can be performed in different ways, depending on versioning strategy.

One of the articles mentioned in the beginning of this chapter, (Feiler 1991) divides SCM models into the following four categories[1].

1. **Checkout / checkin**
   Versions are transferred between the repository (where the version data is kept) and the workspace (where you work on the data).

2. **Composition**
   The composition model supports version selection through rules and assists the user in selecting consistent combinations of conponent versions.

3. **Long transaction**
   The long transaction model where the user connects to a long lasting transaction and operates on a configuration version.

4. **Change set**
   The change set model describes a configuration in terms of change sets, each of which aggregates all modifications performed in response to some change request.

[1]Note that this is in the field of software configuration management (SCM) which is more or less versioning applied to software components in the form of textual source-code, as opposed to for example CAD models in (sometimes) binary form.

4.2 Some versioning terminology

4.2.1 Version

A version is a potentially concrete instance of some object. Therefore, we may have to express which version we are interested in, for example what the features of the version are. Based on this information, a concrete instance may be constructed (if it exists). The version may be constructed in terms of other versions (version oriented) or in terms of the set of features (change oriented).

4.2.2 Revision

The oldest and simplest way to create new versions is to make each new version a modification to the most recent one in a sequential manner. This way, the versions form a single, linked list. This list is what is usually called a *revision chain*. Revisions can have different meanings (Tichy 1988) but I will not differentiate between them in my thesis.

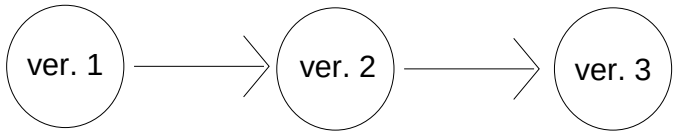

Figure 4.1: An example of a possible chain of revisions.

4.2.3 Variant

Variants, as opposed to revisions, provide you with more than one alternative for the "current" version. Instead of replacing the object like the revision, it provides you with more alternatives in a parallell fashion. There are good reasons for wanting to use temporary variants instead of revisions. These are taken from (Tichy 1988):

- **Temporary fixes**
  A small change is needed to a revision which is not the latest one. We will need to branch off a variant of this earlier revision, so that we don't disrupt the later ones. The change may later be incorporated into a newer revision through merging.

- **Special modifications**
  Local modifications by customers or users should be kept separate from the main branch of development.

- **Simultaneous development**
  Occasionally, two ore more workers may be doing changes to the same objects, changes that are supposed to end up in one common revision later. While the updates are under way, variants will have to be used. This situation can happen by accident, as one can't always know in advance which objects need to be modified, and when. It can also have been planned deliberately, to save time. It is often better to go ahead with two partially independent revisions, than for one to have to wait for the other one to finish first.

- **Parallel exploration**
  When one can't decide in advance which alternative design or implementation strategy is best, a possible solution may be to follow both paths, and then keep the best one. While these alternatives are being followed, variants will distinguish them.

Revisions and variants are usually combined into a common structure called the version graph (see section 4.2.6).

4.2.4 Configuration

A configuration is a composed object. It is composed of a complete and consistent collection of objects with respect to a particular criterion. We may have other types of composite objects as well, but configurations are created to serve a special purpose; it is not a random collection. For example, a particular instalment of a software package may be referred to as a configuration.

4.2.5 Merging

Merging comes into play when we want to splice two or more variants into a common revision. Note that it is not always desirable to merge variants. For instance, to merge an English and a Finnish-Ugrian variant of a manual makes poor sense. There are different ways to do this merge, but in most cases it is an area where computer-aided help is scarce, and human intervention is required. Only in the simplest text-merge situations and in a few other special cases do we have algorithms which can do this task automatically with a guaranteed correct result.

In (Conradi and Westfechtel, 1998) the different approaches to merging are classified as (note that these classes are not mutually exclusive):

- **Raw merge**
  Raw merging applies a change in a different context. For instance, if change c2 was performed independently of change c1, but later

combined with c1, we have a raw merge. Raw merging was first supported by the classic document versioning tool SCCS (Rochkind 1975).

- **2-Way merge**
  The 2-way merge takes two different versions and merges them into a single version. If there are any differences, the algorithm has no foundation for making a decision, and a manual guidance is needed to select the appropriate alternative.

- **3-way merge**
  The 3-way merge is the natural extension of the 2-way merge. In addition to the two versions we want to merge, there is a common ancestor, called a baseline which must be taken into account. This is very useful in deciding how the merged version should look. Generally speaking, if we are merging two textfiles with line-granularity (that is, compared line by line), we have the following rules: If both lines in both versions match the baseline, we have no problem. But if a line is the same in the baseline and only one of the variants, we would keep the edited one in the second variant. If a line is different in all three versions there is no way of deciding automatically, and we have to decide manually. Still this is a major improvement over the 2-way merge.

- **Operation-based merge**
  Operation based merging (Lippe and van Oosterom, 1992) takes two sequences of change operations and combines them into a single sequence, detecting both inconsistencies and conflicts. The application of this algorithm is complex because it generates a huge search space of potentially merged operation sequences which all must be considered. Because of its operation/change oriented bias, it is applicable to change oriented versioning (Munch 1996), see section 4.2.9.

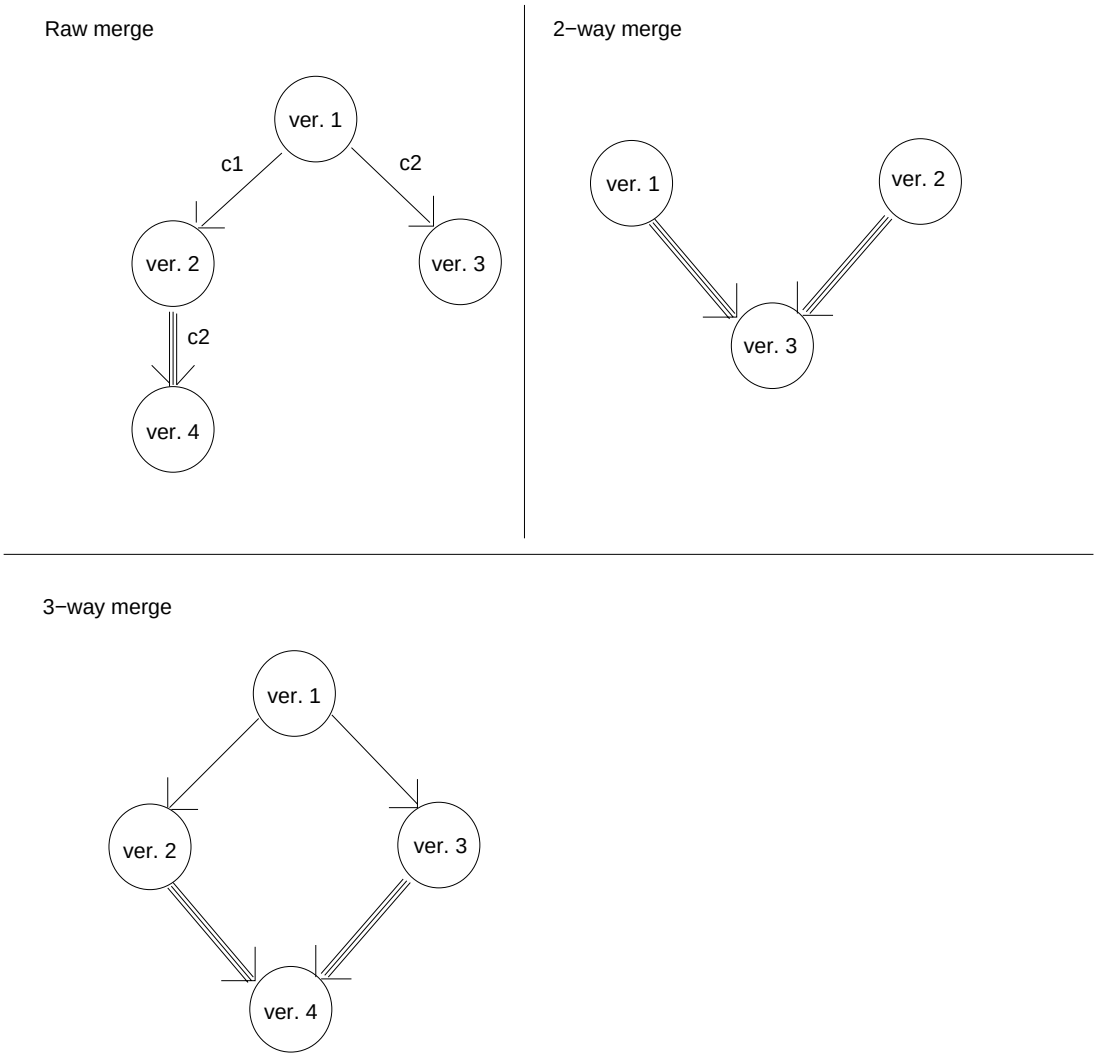

Figure 4.2: Different approaches to merging.

If we have some additional knowledge about what kind of components we are merging, which is called product space in SCM, we can get some additional help from the merge-tool:

- **Textual merge**
  Textual merge can be applied when we know that what we are merging are text files (Adams et al 1986), and can in some cases even be used to merge program code. It sounds like a very weak merge, but gives good results in practice (Leblang 1994).

- **Syntactic merge**
  Syntactic merges can be made with versions of program code, where we have the syntactic rules of the language in a repository, given for instance in extended Backus-Naur form (EBNF). Its goal is to yield syntactically correct merges, but has been realised in very few research prototypes (Buffenbarger 1995, Westfechtel 1991). In addition, it is a big obstacle to expand this form of merge from context-free to context-sensitive uses of the individual programming language.

- **Semantic merge**
  A semantic merge takes the semantics of the language into account. However, it is very hard to come up with a definition of semantic conflict which is neither too strong nor too weak (and is decidable). Semantic mergetools have so far not been implemented with advanced programming languages such as C or C++ but only with simple languages used for research purposes. Good sources on semantic merging are (Berzins 1994, 1995, Binkley et al 1995, Horwitz et al 1989).

4.2.6 Version graph

This version graph is adopted from Munch (1993). The arrows point forward in time and development progression.

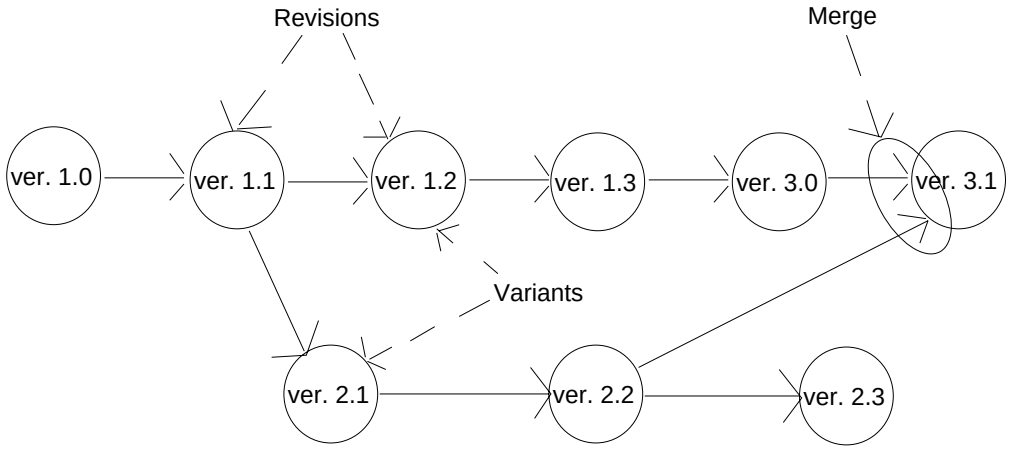

Figure 4.3: An example of a version graph.

4.2.7 Deltas

An informal description of a delta could be that it is the difference (change) between two versions. Conradi and Westfechtel (1998) divide deltas into the following subgroups:

1. **Directed deltas**
   Using directed deltas (Tichy 1982), a version is constructed by applying a sequence of changes to some base version. These changes come as a sequence of elementary change operations.

2. **Symmetric deltas**
   A symmetric delta between two versions, v1 and v2, consists of the properties[2] (v1 \ v2) ∪ (v2 \ v1). In practice this is called embedded deltas and all versions are stored in an overlapping manner so that common fragments are shared. Either each version points to its fragments (Fraser and Myers 1986), or the fragments have control expressions for determining in which versions they are visible. This is called interleaved deltas (Rochkind 1975, Leblang and McLean 1985).

There are many different ways to compute deltas, and a survey of the efficiency differences between *diff, bdiff* and *vdelta* is given in (Hunt et al 1998). The conclusion is that both bdiff and vdelta are far superiour to the old UNIX diff, and that in most cases vdelta is a notch above bdiff.

While it is one thing to compute the diffs, it is quite another to store them efficiently. A recent example of diff storage format is the generic vcdiff by Korn and Vo (2001) which is designed to be an improvement over the standard vdelta storage format.

4.2.8 Version granularity and delta granularity

Version granularity refers to the size of a version and delta granularity refers to the size of those units in terms of which deltas are calculated. In the text versioning tool RCS, for instance, version granularity is at the level of text files, and delta granularity is at the level of text lines.

4.2.9 State-based versus change-based

State-based is the traditional delta-oriented approach to versioning (in some literature this is called version oriented versioning (VoV)). This is regardless of the way the deltas are calculated, and the way they are applied. Change oriented versioning (CoV), is another paradigm, and is

[2] \ is set minus.

based on recording changes. When using the CoV model, we do not refer to a specific version number, but instead refer to a version as the result from applying a specified change-set to a database of objects. CoV has its formal foundation in the article (Lie et al 1989) and is implemented in the EPOS-project, among others. Although state-based is the most widely implemented and used model, and the more explored and easier to implement than CoV, most of the recent version models are in some way or other based on change-oriented concepts.

4.2.10 Extensional versus intensional

Extensional versioning is the classical way of versioning. All versioned objects have a unique identifying version number. Typical use of an extensional versioning environment includes the following: All versioned files must at some point be explicitly checked into the version space. When working on a version, the object (v1) must be cheked out according to its identifier. Later the object is checked back into the object base, forming a new version (v+1).

Intensional versioning does not enumerate its members. They are instead defined by a predicate. A version base is therefore constructed by applying the predicate to specific attribute-values offered by the files in the version space.

In general, extensional concepts are closely related to state based versioning, and intensional ideas are equally tied to the change oriented model.

4.3 Good reasons for versioning

The following reasons for utilizing version management are thought to be of a general nature. However, in section 4.2.3, there are some more specific reasons for allowing variants in particular.

- **Backtracking**
  While in the process of implementing upgrades, we must be able retrieve a product "untouched" by this unfinished upgrade. And if a particular modification was a failure, we should be able to go back to a previous version and start again. This is where one would use *revisions*.

- **Exploration**
  We should be able to explore the history of the evolution of a particular object

- **Exploratory development**
  Since authors can depend on the ability to revert to a known, "safe" state of the system, versioning supports exploratory changes, where the final impact is initially unknown.

- **Comparison**
  It is sometimes desirable to compare two or more versions to see what has changed between them.

- **Safety I** When a product has been delivered to a customer, or is being used in other ways, we must be able to recontruct an exactly identical copy later, either for delivery to another customer, or in order to track down problems. If the customer reports a bug, we can't be sure to find and correct it if we can't rebuild the version that was delivered to this particular customer.

- **Safety II**
  It is desirable to keep snapshots of software objects during various stages of development. This is particularly so when major upgrades are under progress and we still need more stable versions for delivery to customers or for internal use.

- **Rationale capture**
  Since the reason for making a particular change soon fades from memory, versioning systems should allow a brief comment to be associated with each change to capture this rationale. Over time, these comments create a group memory for the object.

- **Reuse**
  By preserving a specific version of an object, the entire object or parts of it may be reused by others.

- **Versatility**
  There is not neccessarily one canonical "current" version of any product - we may want to have, for example, variations for different platforms, languages, or adaptations for different customers. This is catered to by using *variants*.

4.3.1 Reflections around reasons for versioning

Today it is inconcieveable to make a version management tool without some form of database support for the versioned data. This is where Xymphonic collaboration and versioning can form a powerful form of symbiosis. The Xymphonic model, with its simplicity and generality, seems to be extendable by versioning. This is the topic of the next chapter in the thesis.

4.4 Some version models

Versioning had its beginning in 1975 when Mark Rochkind developed SCCS, the Source Code Control System (Rochkind 1975). SCCS relies on interleaved delta storage for text files. It is also based on state based versioning and uses the well known check-in, check-out model.

RCS (Tichy 1982, 1985) is a successor of SCCS, and it uses directed deltas instead of the interleaved deltas we find in SCCS. RCS stores the latest revision without deltas because, by experience, this is where we usually want to continue working. If we want to continue work on another version, it will be computed using backward deltas (and forward if it is on another branch). RCS is one of the older version control systems. Due to this, it has played an imporant role in shaping ideas and strategies of later systems. For example, a widely used version system such as the concurrent version system (CVS) (Grune 1986) is based on RCS ideas and storage strategy.

An early example of change based versioning is the PIE system (Goldstein & Bobrow 1980). Later examples include Aide-de-Camp (Cronk 1992, Software maintenance and development systems 1990) and EPOS (Lie et al 1989, Munch et al 1993).

Later systems such as ClearCase (Leblang 1994) and ICE (Zeller & Snelting 1995) support a virtual file system to enable smooth tool integration. Zeller and Snelting (1997) explore a long transaction model in conjunction with ICE. However, this does not compare well to version extended Xymphonic transactions because the long transactions ICE uses are of a more liberal kind that those from database theory.

Conradi and Westfechtel (1998) claim that version models are converging to an increasing extent. They also believe that a version model which integrates extensional and intensional versioning, state based and change based versioning, revisions, variants and derived versions can be distilled into a coherent framework. This framework should be customizable to suit the needs of specific applications, that is, it should be application oriented.

4.5 Areas well suited for versioning

Today, most projects are suitable for versioning, including projects where only very few people are cooperating. As projects grow larger, the need for good versioning tools increase rapidly.

The kinds of projects which can benefit the most from versioning include:

- SCM systems, which include programming projects and regular text document cooperation.
- Hypertext systems (Whitehead 1997), which may be similar in some respects to any structured (technical) documentation.
- CAD systems, which is discussed in (Katz 1990), and (Dart 1992)

4.6 Chapter summary and conclusions

In the course of this chapter a brief explanation of central concepts in the field of version management has been given. Definitions have been given for important terms such as configurations, revisions, variants, deltas, merging and more. In addition, some version models have been touched upon and motivations for applying versioning concepts to different fields of computer use have been listed.

Chapter 5

A Xymphonic model for versioning

There is no single, easy answer to the question of what the desired behavior for a version model is. While we would like to see as many nice features incorporated as possible, it is important to keep the design sleek and not stray too much from the original path, which was to see in what way the Xymphonic model could conceptually be combined with concepts from the field of versioning.

5.1 Introduction

This section introduces some new terms particular to this model. Some definitions of transaction and version management will also be given in order to provide a starting point for motivating a combination of the Xymphonic model with versioning.

Anfindsen (1997) states the following on page 23:

> A transaction model is a specification of allowable and mandatory behavior for transactions as well as their structure.

Of course this allowable and mandatory behavior should preferably give us some attractive properties. For the Xymphonic model, these properties encompass controlled access to the resources in the DB, resulting in serializable histories by a choice of Xymphonic correctness criteria. What is normally understood by Xymphonic transactions is a transaction history serializable by the NCCSR criterion. This gives users the possibility to communicate, for example the maturity of data and also to browse each others work in a controlled manner, solving the problem of read/write and write/read problems. Also, users can choose to

convert write locked resources to Xymphonies and thus delegate recursively and share write access with other users, solving write/write problems. Everything seems brilliant, so why bother this design with versioning features? To see why versions are indispensible in most cases of long lasting cooperation, consider the following highlights from chapter 4:

- **Backtracking** when we need to go back to an earlier version
- **Exploration** of the evolution of an item
- **Exploratory development** is possible when we know we can revert to a safe state
- **Comparison** is possible if we want to see what has changed from one version to the next
- **Safety** in the sense that we can go back to a particular version and fix errors there when they are discovered
- **Safety** in the sense that we have versions known to be stable when a system is undergoing big changes
- **Rationale capture** by writing comments for each new version
- **Reuse** by being able to use suitable versions of objects in other projects
- **Temporary fixes** to older revisions
- **Special modifications** which are not natural in the main line
- **Parallel exploration** when we do not know which approach is best
- **Versatility** when we need different variants for language, operating system, etc.

Note that savepoints may to some extent be regarded as versions; however, they are *not* comparable to this model because savepoints are only manufactured in order to provide for transaction recovery in case of failure, while the versions of resources manufactured by version extended Xymphonic cooperation are designed specifically to last beyond the duration of the transaction (but only if it commits!).

The offset from the Xymphonic model to this extended model is the invention of some new lockmodes, the version write locks VAR and REV, the immutable version locks Ω_{VAR} and Ω_{REV}, and a versioning unlock lock, the blank lock (BL). These new locks and their compatibilities with

the traditional locks are explained in section 5.3 in particular, and the rest of this chapter in general. To provide for the manufacturing and storage of versions, a version SOC (VSOC) will be introduced alongside a nested version repository.

In (Kim 1995, 414), Gail Kaiser makes an argument about the neccessity of supporting configurations of resources. She states (added comments in square brackets):

> The key omission**[**of common traditional versioning**]** is not keeping track of which versions of objects are consistent with each other. For example, if each component (object) of a program has multiple versions, it would be impossible to find out which versions of the components actually contributed to producing a particular executable that is being tested. It is neccessary to group sets of versions that are consistent with each other or otherwise together into *configurations*.

This model presents an outline of how management of configuration versions can be carried out. This will be explained in more detail in section 5.5.

As a final note in this introduction, a quotation from Anfindsen (1997, 104-105) about CCSR in relation to multiversion concurrency control (MCC) is given:

> However, some interesting combinations of CCSR and MCC appear to be possible. Consider e.g. a designer who is in control of an LLT and is about to perform a number of related updates. If this designer could carry out those updates in a separate subtransaction that would create short-lived versions of the objects in question, parametrized readers accessing those objects could be protected from seeing inconsistencies due to work in the subtransactions being only partially completed. That is, I believe it would be possible to combine CCSR and MCC in such a way that related but uncommited updates could be disclosed atomically, and that such access to the previous version of the data items in question is possible while the new version is under construction.

From this argument it would seem that Anfindsen would like the versions under construction to be delegated to a subtransaction and use a special type of parameter set. The version mangement in question is a subset of the functionality embodied in the modest extensions described in this chapter. As a result, offering the functionality desired,

amounts to creating a lock parameter for this special case. An example of this will be given in section 5.4.8.

5.2 Group types

Group type implies any part of the model which may logically contain or reference other parts. Based on that criterion, we have the following group types:

- **Transaction**
  Transactions may control WSOCs, RSOCs, Xymphonies and an arbitrary nesting of subtransactions as specified in the Xymphonic model. For versioning, Xymphonic transactions must also control VSOCs and version repositories.
- **WSOC**
  A data sphere containing write locked items.
- **RSOC**
  A data sphere containing read locked items.
- **VSOC**
  A special sphere for version management belonging to a transaction and containing write locked items with versioning features. We say that the elements in the VSOC are version write locked.
- **Version repository**
  A version repository holds a set of immutable versions. It can belong to either a transaction or a Xymphony or it can reside in the top level DB. Versions in the repository are locked on a permanent basis with an Ω lock. Versions in repositories owned by transactions are immutable only to the extent that they may not be edited, but still may be aborted and thus deleted in case of transaction abort.
- **Xymphony**
  A Xymphony is used to denote a collection of resources held by a transaction in the system by using a Xymphony lock. Other users may join and participate in the Xymphony as described earlier in chapter 3.
- **Resource**
  The generic resource data type may reference other resources, for example as version parent/child. Note that what will be referred to as a resource throughout this chapter may be any type of data element, e.g. a file, a relational table, tuple, etc.

- **Configuration**
  Configurations may reference both resources and other configurations.

5.2.1 The VSOC

A transaction needs a VSOC to keep track of the current versions in production. The VSOC thus holds resources and configurations with version write locks, being either VAR or REV. Only transactions can be holding VSOCs.

5.2.2 The version repository

A transaction needs a version repository to keep track of its yield of immutable versions. Each Xymphony, on the other hand, needs a version repository to keep track of the new versions its transactions turn over when they commit.

An important notice is that only subtransactions may create version successors of resources located in a repository owned by a transaction. If this were not the case, we could get into a situation where an independent transaction is working on a version successor and the transaction holding the predecessor aborts. This will lead to inconsistencies and is clearly undesirable. However, if only subtransactions can create successors, we know that they will either commit or abort before the ancestor transaction and ergo no version inconsistencies will be created.

5.2.3 An example

Figure 5.1 shows a legal nesting of group types. The transaction T_{Alice_a} has a VSOC and a version repository in addition to the traditional Xymphonic elements. Also note that the top level DB has a version repository and that the Xymphony belonging to T_{Alice_a} also has a local repository. Bob's subtransaction $T_{Bob_{aa}}$ is working on some versioned resources within the Xymphony and is accordingly holding a VSOC. When this transaction commits, the new versions will be commited to the version repository of the Xymphony.

5.3 Locking

In this section, the traditional lock matrix as described in section 3.3, is extended by adding new locks for versioning. Immutable versions are locked with Ω locks. We may create a child version of an Ω locked

item by VAR or REV locking it. A more detailed description of the new lockmodes will be given below.

The Ω_{VAR} lock

The Ω_{VAR} lock is used in order to mark an immutable resource version for variant semantics. That is, resource versions locked with this lock allow more than one version child to be created. If an unversioned resource is locked with this type of lock, it is in effect moved to the version repository of the acting transaction, made immutable, and marked for variant semantics. This is provided that the acting transaction is holding a write lock or stronger on the resource.

If a resource is VAR or REV locked, then only the transaction holding the lock may apply an Ω_{VAR} lock. Doing this means commiting a new version to the local version repository of the transaction. The version number of the version write locked resource is accordingly incremented and the user can continue to work here until the transaction is either commited or aborted. Informally, this equals taking a version snapshot of the resource.

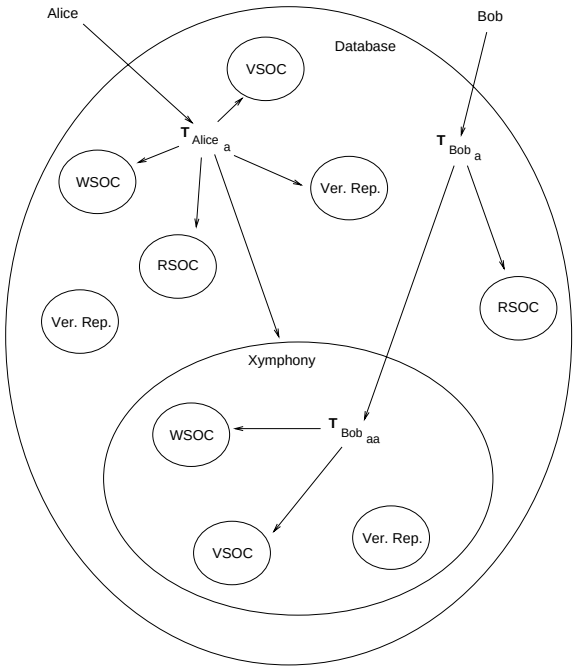

Figure 5.1: An example of nesting

Extended lock matrix											
	Granted lock										
Requested lock	Ω_{VAR}	Ω_{REV}	BL	B	R	U	W	VAR	REV	Xymphony	X
Ω_{VAR}		×									
Ω_{REV}	×										
BL	×	×									
B	×	×	×	×	×	×	×	×	×	×	
R	×	×		×	×	×					
U	×*	×*		×	×						
W				×							
VAR	×	×		×							
REV	×	×		×							
Xymphony				×							
X	×**	×**									

Figure 5.2: Versioning extended lock matrix.

Another situation occurs if a resource is locked to a repository with an Ω_{REV} lock and a user desires to branch it out into variants. One solution is to apply an Ω_{VAR} lock to the version. Doing this results in a version child of the Ω_{REV} locked resource, to be commited to the version repository of the transaction with an Ω_{VAR} lock. This kind of action effectively overrides the versioning semantics of the old version and should therefore probably be restricted to e.g. special situations or privileged users.

Because resources residing in repositories are immutable, granting read locks is unproblematic. Note that the version being read in this fashion may be an old copy as other transactions can have version write locked successor versions. This problem of not getting the most current data is one of the drawbacks of MCC as pointed out by Anfindsen (1997, 104). However, because versioning is not used as a means of concurrency control in this model, it will not be a problem as long as the user realizes that the data may be stale.

The Ω_{REV} lock

It and the Ω_{VAR} lock have the bulk of their compatibilities and properties in common, except for the fact that Ω_{REV} locks are used to mark versions of resources and configurations for revision-only semantics.

If a transaction has the access privileges to reach an Ω_{REV} locked resource version, and the version does not already have a successor, the transaction may version write lock the resource. Because revisions may only have one child, this cuts off other transactions from creating immediate child versions.

The BL lock

The BL lock is used to remove a resource from version control[1]. That is, if a resource resides in a version repository and we would like the successor of it to be removed from versioning and rather work on it through standard Xymphonic procedures, we must use the BL lock. Upon doing so, it could for instance be recorded that the resource was removed from version management at the specified time by a particular actor for a particular reason. Using the BL lock and removing a resource from versioning can have drastic consequences, ergo should this lock probably be restricted to particular users/usergroups or special sitations.

Upon locking a resource with a BL lock, the transaction owner signals a desire to write. Accordingly the BL lock has the same compatibilities as a normal W lock. Note that in order to be able to set a BL lock in the first place, the version must be held in a repository that the pertaining transaction has access to. These can be either the top-level repository residing in the DB, a repository belonging to the particular transaction or an ancestor transaction, or a repository belonging to a Xymphony where this transaction participates.

On the other hand, if the resource version resides in a VSOC, a BL lock can not be granted because this means the resource is either VAR or REV locked, and that we are in the middle of producing a brand new version. However, after the version has been commited to a repository, it will be Ω locked and a BL lock will be granted if the pertinent repository can be reached.

The VAR and REV locks

The lock compatibilities of the VAR and REV locks are the same as those of the write lock because we are in fact writing to a new version. When an Ω locked version is locked by one of these locks, a mutable successor is created, provided that the lock is granted. If it is granted, the version write locked item is located in the VSOC of the transaction.

If the transaction is later aborted, the new version is erased. On the other hand, by commiting the transaction it is implicitly signalled that the version is ripe for commitment and it is then Ω locked to the version repository of the parent transaction, enclosing Xymphony or top level DB.

[1] It is not applicable to configurations because they are control structures particular to version control.

The X lock

On rare occasions, we may like to delete immutable versions, see for example section 5.4.8 on combining CCSR with MCC. In order to do this, we must lock the affected items with the X lock to make sure there is no interference with the operation. Removing a version is a drastic thing to do, and should probably be restricted by some criterion like the use of the BL lock.

Comments to the lock matrix

* Upgrade mode is granted for Ω locked versions, but may only be upgraded to either VAR or REV.

** In certain situations, we may want to delete an immutable version. In order to do this, the version must be X locked.

Some remarks on versioning semantics

It is not a trivial task to decide what the outcome should be if a transaction performs one of the following actions:

1. Requsting a VAR lock on an Ω_{REV} locked resource.
2. Requsting a REV lock on an Ω_{VAR} locked resource.
3. Requsting an Ω_{VAR} lock on a REV locked resource.
4. Requsting an Ω_{REV} lock on a VAR locked resource.

All these four actions change the versioning semantics of a resource, and should probably be restricted by some yet undefined criterion. This problem will not be treated in this thesis, but it is noted as an item which needs to be looked into, and is accordingly mentioned in the chapter on further work, accompanied by some thoughts on possible solutions.

5.4 Versioning of resources

Data elements will be referred to as resources. Examples of resources are files of sourcecode, a method in a file of sourcecode, a CAD model, a table in a relational database, a tuple, etc. In software configuration management (SCM), the predominant resource and version granularity is at the file-level. Configurations will not be treated as data elements.

Resources can be of different granularities. For instance, for a versioned

filesystem, such as for example DSEE/ClearCase (Leblang 1994) or ICE (Zeller & Snelting 1995, Zeller 1996), some resources will be the leaves, typically the files, in a tree of resources. Other resources will be the inner nodes, in this case the directories, of the corresponding tree. For a relational DBMS this form of hierarchy could, for example, be table → tuple → attribute.

The motivation for differentiating between inner nodes and leaves is that, trivially, when we get to a resource leaf, we can either lock and work on the whole of it, or not. This may be a waste of opportunity for collaboration. Consider the situation where a piece of data has evolved over time and accumulated a lot of data. This can, for example, be a file of source code or a document which has evolved. There may be a number of people interested in writing to this file, but only one may write at a time.

A current solution is to make the file available to other participants through a Xymphony. This is fine, but still only one may write at a time. What we would really like is to be able to dynamically split and merge these leaf resources in a controlled manner, avoiding optimistic concurrency and the problems of merging overlapping updates. This is not given a comprehensive treatment in this thesis, but an outline of a solution is given in section 5.4.7 and some ideas for further work is given in the next chapter.

Note that when granularity is changed this way, a leaf will, at least for a certain duration of time, no longer be a leaf, but instead in inner node. However, it would still be nice to identify its "true" place in the hierarchy. To meet this end, leaf resources will not be referred to as leaves, but instead as "outer" resources. This avoids any potential conflicts with the normal meaning of leaves as the extremities of a tree. It is thought that an outer resource may reference other outer resources. More on this in the next section.

5.4.1 Relations between resources

In order to make a model of the relations, shown in figure 5.3, Nijssen Information Analysis Model (NIAM), (Nijssen 1981) was used. An important notice is that the figure is only intended to show the generic relations which are important for the management of versions. Of course, in real life we would like a version of a resource to have a timestamp, user comments, perhaps an easily identifiable alias, etc.

Resources can be in the following many-to-many relations:

- **Reference list**:
  This relation can mean two things based on the value of the type attribute. If we have either inner→inner or inner→outer relations we have a relation of hierarchic ownership, but if we have the outer→outer relation it means we have a situation where one outer resource owns another and is (partly) composed of it, i.e. the granularity is changed.

- **Version list**:
  This relation gives a list of the edges between resources in the version graph.

The type attribute can have two values, either "inner" or "outer" as explained in the beginning of this section. Note that if we do not desire the ability to change granularity dynamically, the attribute has no function and should be omitted.

Not all combinations of connections in the reference list between resources with different type attributes are permissible, and they all carry different meanings:

- **Inner → inner**:
  This can for instance be the parent/child relation between two directories.

- **Inner → outer**:
  As an extension of the former item, this could be the parent/child

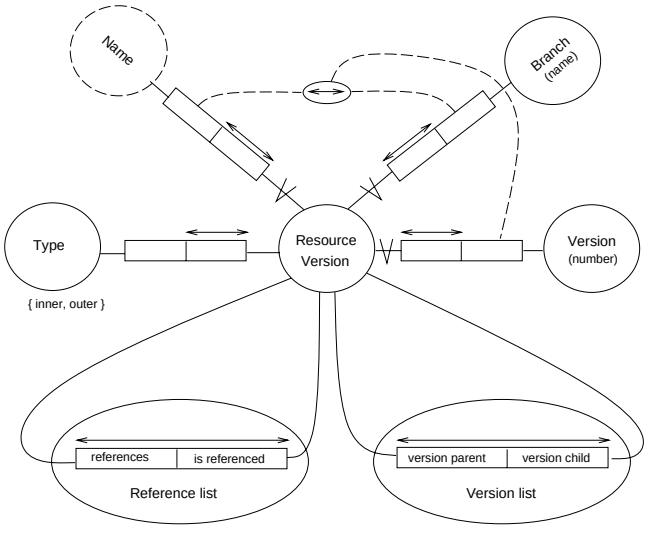

Figure 5.3: NIAM model of resource relations.

relation between a directory and a file.

- **Outer → outer**:
  This denotes that a resource is composed of editable fragments and should occur when the granularity of a resource is changed.

- **Outer → inner**:
  This is impermissible.

5.4.2 Identifying versions

In a DBMS setting, it could for example be possible to run queries on the attributes of versions. In most cases, one would want to continue work on the version of a resource satisfying a particular predicate. If queries are availiable, it would mean that one is able to identify versions both intensionallly (by predicate) and extensionally (through explicit enumeration).

It must be decided how it can be identified that a version is a variant of another version. With the suggested relations between versions of resources, some additional attributes would suffice. ClearCase (Leblang 1994), for example, assigns names to the edges of the version graph: `mysocket.c@@/main/test` denotes a path in the version graph of the resource `mysocket.c`. In this particular case, the path identifies this variant of `mysocket.c` to belong in the test branch parallell to the main branch.

Another scheme used by e.g. Adele (Estublier & Casallas 1994) and the attributed file system of SHAPE (Mahler 1994), is to create *attribute/value* pairs. For example, one variant may have an attribute, say, language set to English, while another has the adherent value of Norwegian for the attribute. Considering our attribute oriented definition bias, this seems like a good solution in the case of our model and will be employed in the implementation chapter, appendix A in the thesis.

A consequence of this is that it may not be possible to uniquely identify a resource version by version number alone. Consider the version graph in figure 5.4. It is commonsensical that when requesting version 4 of `manual.pdf`, we will get a set containing the two variants in return. These can then be separated on the basis of the value of the language attribute.

In any case, a combination of resource identifier/name (e.g. `manual.pdf`), version number and a variant identifier should be sufficient to uniquely

identify any version.

Also note that versioning systems such as RCS (Tichy 1982, 1985) and SCCS (Rochkind 1975) use numbering levels to identify temporary variants, e.g. `1.1` and `1.2` are variants stemming from 1. This could seem useful for a solution of combining CCSR and MCC, but because this will be solved by using attributed variant identifiers, tailoring the scheme to Xymphonic versioning will not be attempted.

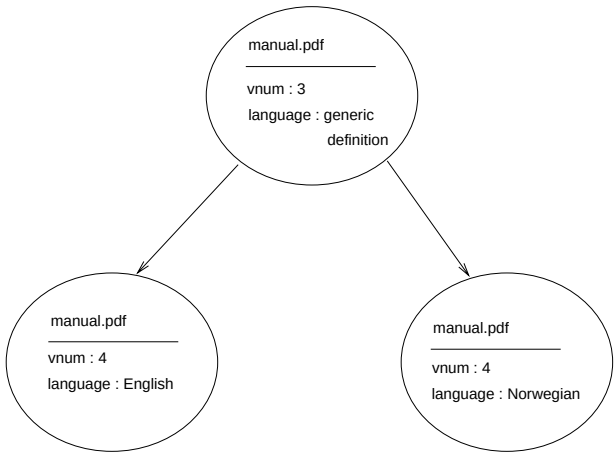

Figure 5.4: An example of variant versions with branch identifiers.

5.4.3 Selecting a version

Selecting a version of a resource for work is a two stage operation: First we must identify the version we would like to work on, then we must attempt to lock the resource with the desired lock. In order to be able to create a new version, the following conditions must be met:

1. We must be able to reach the old version of the resource, that is, it must reside in a version repository within the reach of our transaction. These are:
   - The repository of the top level DB.
   - The repository of our transaction or an ancestor transaction.
   - The repository of a Xymphony our transaction has access to.
2. The desired lockmode must be granted.
3. If the old version is locked in Ω_{REV} mode (remember that Ω locks sticks for eternity), a successor can not already have been created.

4. If the old version is locked in Ω_{VAR} mode, a successor can not already have been created on the branch we wish to continue.

If all these tests are successful, we are granted a version write lock on the successor resource which we can work on in the VSOC of our transaction. However, setting a BL lock will remove the versioning status of the child version[2] and give standard W lock and unversioned resource semantics. This unversioned resource does accordingly not belong in the VSOC, but rather in the WSOC of the transaction.

If our desire is to acquire a read lock on the version, we do not have to care about items 3 and 4 in the enumeration, but the other two conditions must be satisfied. A special case occurs if we wish to read the contents of an Ω locked resource, we only have to care about the first item, being able to reach the version. This is based on the fact that Ω locked versions are immutable, i.e. they may *never* be edited again. Accordingly, there is no chance of a read/write conflict.

5.4.4 Creating new versions

Once a version has been commited to a version repository, it is per definition immutable and can itself never be edited again[3]. When we want to continue on the next version by selecting it and getting the appropriate version write lock, we get a mutable copy, a version child to begin working on. An example of creating versions is given below. It begins by version locking a resource in the version repository of the top level DB.

The initial setup can be seen in situation 1 in figure 5.5. First we try to version write lock version 3 of the resource `rs`, denoted as RS_3. Supposing this went ok, the version child reference from RS_3 is by some means marked as occupied if it is Ω_{REV} locked. How this is done is considered implementation specific, but it should be kept in mind that the operation must be rolled back in case of transaction abort. On the other hand, if RS_3 is Ω_{VAR} locked, we must record that a child version is in progress on a given branch in order to prevent others from creating versions that would collide on the same branch.

Provided all this went ok, a mutable copy of RS_3, denoted RS_4 with a version write lock is then transferred to the VSOC of our transaction. We may edit it for some time until we decide to do one of the following:

[2] It will *not* remove the stored graph of versions, it will merely create an unversioned successor.

[3] It can, however, be deleted either after X locking it, or if the repository belongs to a transaction or Xymphony which aborts.

1. Record this version, but continue working on the next version child.

2. Allow others to create version children by making it immutable and adding it to a Xymphony.

3. Perform an action is which triggers the creation of a new version.

4. Commit the transaction

5. Abort the transaction

We will discuss the first three possibilities here, and items 4 and 5 will be considered in section 5.4.6.

If we have reached a particular goal or if for some other reason it is a suitable situation for recording a version, we can do this either by commiting the entire transaction, or by locking RS_4 with one of the Ω locks.

If we have other work we would like to commit as well, it might be a suitable action to commit the entire transaction. However, in some cases, this may be viewed as a somewhat drastic measure. If we would like to merely create a new version, but keep editing the successor, RS_5, we would apply an Ω lock. When RS_4 is locked with one of the Ω locks, its contents are made immutable, the user may be prompted for version information, and a timestamp is applied, etc. The immutable RS_4 is then moved to the local version repository of the transaction and, if automated, the successor RS_5 with a version write lock, residing in the VSOC is returned, see situation 2 in figure 5.5. Keep in mind that both RS_4 and RS_5 will be eradicated if the transaction aborts.

If we would like others to contribute to this part of the version graph of the resource, we could make it accessible through a Xymphony. Let us consider the situation where we have RS_4 version write locked in the VSOC. In order to make it available to others, we must in essence make it immutable and commit it to the version repository of the Xymphony. Users invited to the Xymphony may access this repository in the same way as the repository of the top level DB was accessed, see situation 3 in figure 5.5. The resource should be made immutable in this situation because:

- It would not make sense to locate a version write locked resource to a Xymphony, because this means the resource version is write locked to one transaction and thus unavailable for others.

- Converting the VAR/REV lock to a Xymphony lock would remove the versioning semantics and should not be allowed (in this case we should BL lock the version first).
- It must be assumed that the resource in question is in a stable state when it is handed over to the Xymphony. This should be recorded.
- We would like to track one change to one person/transaction.

Events triggering the automatic creation of a new version could, for example, be changing the lock parameter set, a version snapshot could be taken every half hour, etc. There is probably an abundance of possible product-space specific version-creation triggering actions.

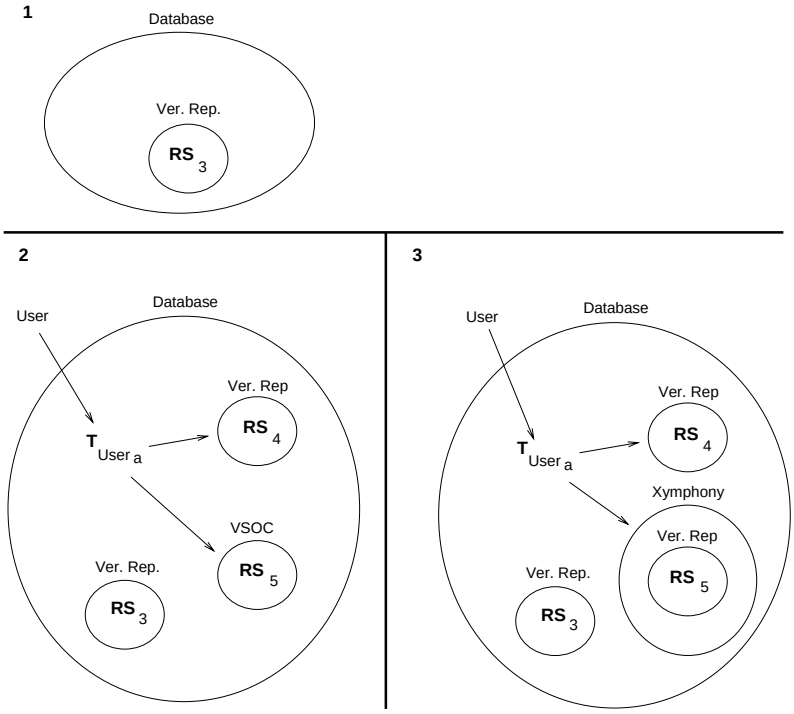

Figure 5.5: Version creation overview, first figure.

5.4.5 Performing merges

In many cases, there can be no question about the usefulness of allowing variants of resources, see esp. section 4.2.3. Once more, it should be stressed that the Xymphonic transaction model caters to all concurrency control. Versions are *not* intended to be used as a means of this.

But for some reasons, also given in section 4.2.3, we would sometimes like to be able to merge variants. This is not a core topic of the thesis, but a solution will be outlined.

We should get version write locks on the versions we want to merge. This will result in a collection of version children locked to the VSOC of our transaction. After the required version write locks have been obtained, a merge process can then be applied to the versions, resulting in a common version child.

Regrettably, this is not an optimal solution because it yields some redundant versions, i.e. the ones we get version write locks on first and then merge will be recorded as new versions although their contents have not been altered. It is highly probable that improvements can be made on this, maybe involving automatic X locking and deletion of the redundant versions. Another possible solution is a version idempotency detector, i.e. if a new version is created which is identical to its parent, it could be discarded.

Why merges can not be performed in a straightforward manner by getting read locks and creating a merge based on the contents we can read deserves an explanation: Unfortunately, that approach will result in a creating brand new resource, which without special considerations will be completely disconnected from its ancestors. Alas, read locks are not intended to automatically produce new version numbers or resource version relationship bonding.

However, it is conceivable that if we notify the immutable parent versions that a child version is on its way as we would do automatically when version write locking as explained in section 5.4.3, the approach with read locks could work. Due to this special notification, we may call the operation, for example, a read-to-merge[4].

5.4.6 Commiting and aborting versions

Both commit and abort of transactions carrying versions will be considered in this section. The two situations considered will be extensions of those depicted in section 5.4.4 and illustrated in figure 5.5.

For the commit of transaction T_{User_a} in the situation where it holds RS_4 in its version repository and RS_5 in its VSOC, the result is shown in situ-

[4]It should probably not be called a read with intent to merge because it is indeed not an intent lock, just a normal read lock with some extra requirements.

ation 2 in figure 5.5. If we should choose to abort this transaction, all of its edits are lost. This includes the contents of its repository and VSOC. We are thus back to square one, situation 1 in figure 5.5. It is also important to roll back the expectations of the version parents, which assume there is a new version child on its way, when it is in fact aborted.

The second situation, where the transaction owns a Xymphony, we shall first consider the commit of the Xymphony. This results in the transaction regaining prior control over the contents of the repository owned by the Xymphony. If no one has been continuing work on the version, we are returned to situation 1 in figure 5.6. Commit in this situation adds the new versions to the repository of the top level DB and is shown in situation 2 of figure 5.6.

If a transaction had been continuing work on the version in the repository of the Xymphony, we would get a situation of nested cooperation. Just like T_{User_a} commits to the repository of the top level DB, transactions in the Xymphony would commit their new versions to its version repository. When the Xymphony is commited, these will be added to the version repository of T_{User_a} which may or may not further choose to commit the data.

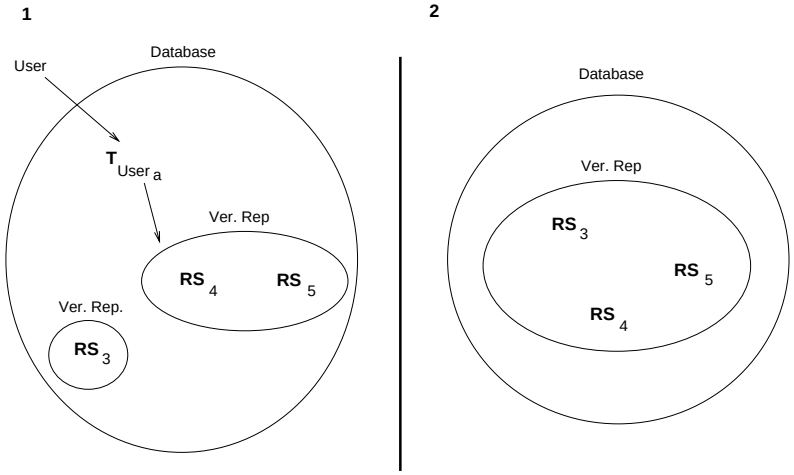

Figure 5.6: Version creation overview, second figure.

If the Xymphony is aborted, on the other hand, the principle is that its results should be rolled back. However, it would be required for T_{User_a} to regain control over the version it commited to the repository of the Xymphony. In our example, this would actually result in the outcome being as shown in situation 1 in figure 5.6.

The described outcome makes sense because, if the Xymphony had not been created, we would instead have commited the version directly to the repository of the transaction. Also note the complete overlap in outcome between commiting a Xymphony where there has been no collaboration, and abortion of the same Xymphony.

5.4.7 Manipulation of versioning granularity

Sometimes, resource and version granularity is at an undesirably high level. Although finding solutions to this in connection with Xymphonic collaboration has been considered interesting work in connection with the other ideas in this thesis, it never made it sufficiently high on the priority list to receive a comprehensvie treatment. Nevertheless, an outline of a possible solution is given below.

One solution could be to allow transactions to create fragments, but only to commit complete versions of resources in respect to the version parent.

An example scenario consists of Alice, VAR locking a version of a resource. Note that when her VAR locked version is commited, either to her local repository by applying an Ω lock, or through commiting her transaction, the Ω locked version parent of her VAR locked version expects to get a suitable version child. By suitable is implied a version child of the same type and granularity, in order to make computation of e.g. delta offset trivial.

Alice may want to split the version into fragments if:

- She decides that it contains too much data, making it unfair for her to keep posession of it entirely.
- Other users explicitly ask her for write access to parts of it.

Whether this fragmentation is performed manually, or due to a specialized description is not the issue here. When a desire to fragment is signalled, the VAR locked resource is Ω locked to the version repository of Alice's transaction and VAR locked fragments are created in her VSOC. These fragments can now be viewed as any normal form of resources and may be made available to others through a Xymphony. Transactions in the Xymphony may choose to fragment the fragments further in a recursive manner.

When other users have commited their work on the fragments, Alice

may be in the situation of figure 5.7 regarding the original version and its fragments.

The fragments were created as, and should still be disjoint segments of the original resource version. This makes it safe to merge the fragments to new versions automatically, probably most intuitive in a breadth-first traversal manner. Let us assume that the resource version Alice originally locked was revision 3. In the following example F (0) $_0$ will mean fragment 0 in version 0. As shown in figure 5.8, this makes { f1.v0 , f2.v0 , f3.v0 , ... } into version 4, { f1.v1 , f2.v0 , f3. v1 } into version 5, { f1.v1 , f2.v0 , f3. v2 } into version 6. This demonstration shows the general idea.

When Alice decides to commit her fragments, a new chain of versions of the resource is commited as expected.

Unresolved issues are creation of variants of fragments and merging fragments. A preliminary idea may be to create variant (complete) versions in the merge process at points where variant fragments occur and, if merged, create merged versions (originating from the fragment variants) where fragments are merged. An example of fragmentation including variants and a merge is shown in figure 5.9, and figure 5.10 has a high level view of the new resource versions.

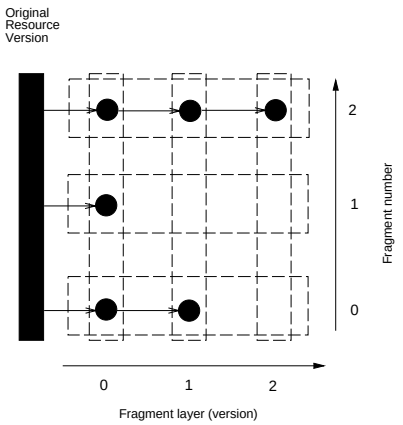

Figure 5.7: Resource after work on fragments.

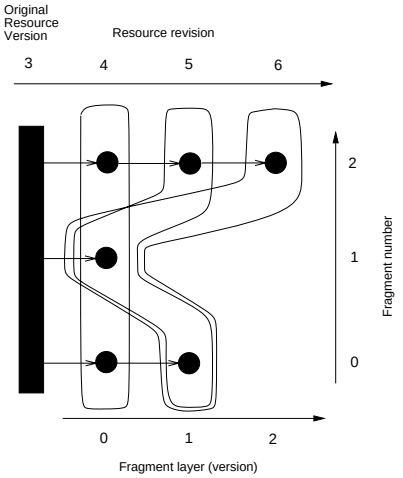

Figure 5.8: Merging fragements.

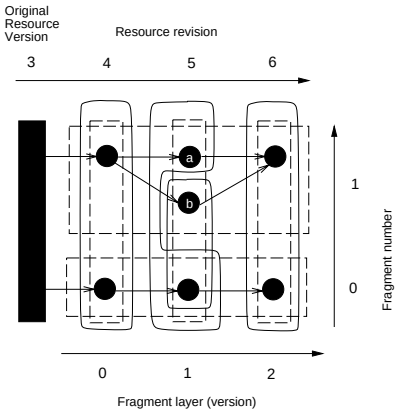

Figure 5.9: Prospective merges in a situation where fragments are in variant and merge relations.

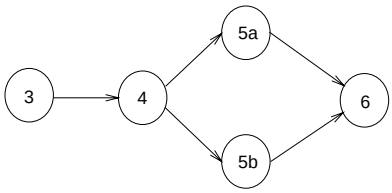

Figure 5.10: High level view of the situation in figure 5.9.

5.4.8 Combining CCSR and MCC

In his PhD thesis (1997), Ole J. Anfindsen argues on pages 104 and 105 that a combination of the CCSR correctness criterion of the Xymphonic model and MCC as found in for example the Oracle DBMS, could be beneficial for short lived updates. The idea is that while one user is performing these updates, others could view an older version of the data in question. This filtration should be performed by the means of parametrized access, and the updates should be created atomically through a subtransaction.

This section rests on an example based on the five steps illustrated in figure 5.11 and outlined below:

1. In step one, T_{User_a} has a write lock on the resource RS but would like to do some quick edits and employ versions to do this.

2. In the second step, T_{User_a} Ω_{REV} locks RS, resulting in the creation of RS version 0 and places it in the local version repository.

3. In step three, T_{User_a} spawns a subtransaction, $T_{User_{aa}}$. The subtransaction then REV locks RS, resulting in the placement of RS version 1 in its VSOC. It will use a special lock parameter, for example "versioned update", "short update", or "unstable". At this time other users may read the Ω locked contents, but because the version is in fact Ω_{REV} locked they may not create version children.

4. In step four, the subtransaction commits, adding version 1 of RS to the version reposotory of T_{User_a}. The short update is finished; we must now tie some loose ends.

5. In the fifth step, the version commited by the subtransaction is BL locked, resulting in a fresh, updated RS placed in the WSOC. In order to clean up after this MCC operation, the transaction may want to X lock the versions in the repository and delete them.

There is probably room for improvements here. For example, some steps should be combined into atomic actions. In particular step 2 and 3 should be combined into an atomic unit to ensure that no other transaction snatches the version child reference from the Ω_{REV} locked version. Also, the cleanup in steps 4 and 5 should be combined into an atomic unit to avoid other transactions read locking the new version and thereby creating a conflict with the BL lock we want to set.

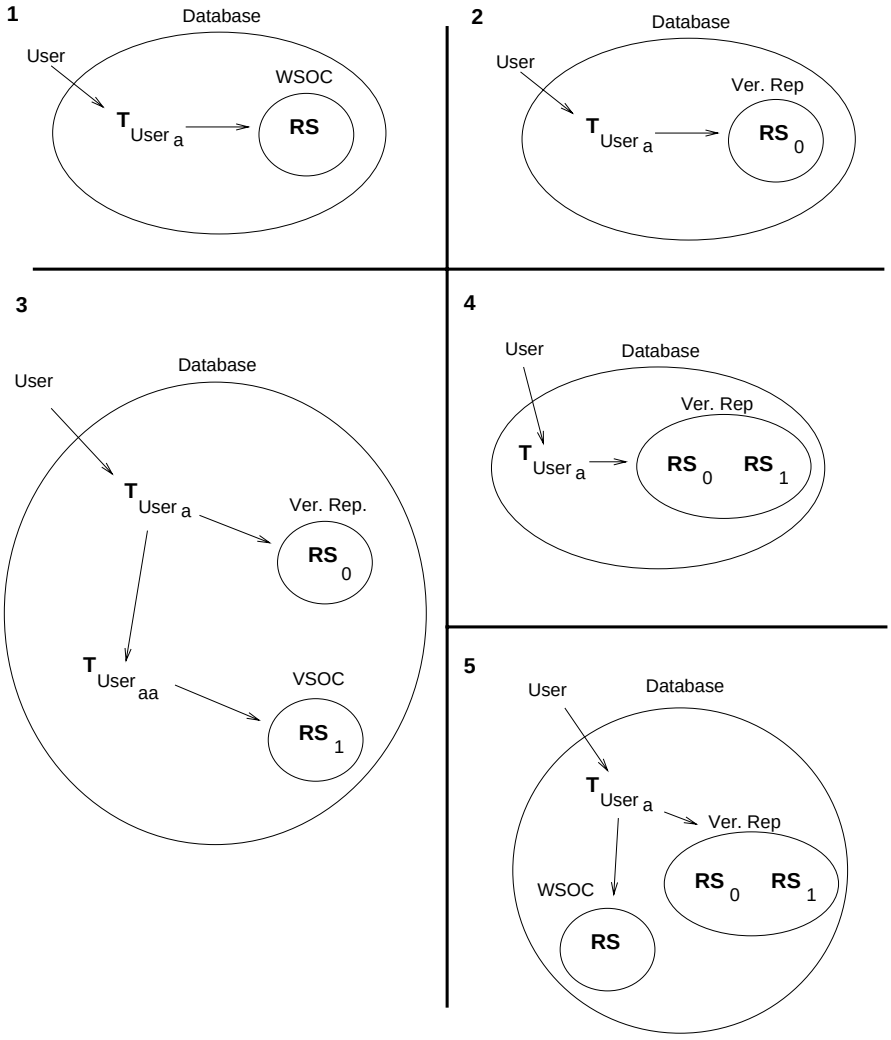

Figure 5.11: An example of combined CCSR and MCC.

5.4.9 Resource versioning summary

The resource datatype denotes an arbitrary type of data item. It can, for example, be a file, a table, a tuple or a persistent Java object. This part of the model chapter has shown how to create revisions and variants of resources by using new types of locks.

These new locks are the version write locks VAR and REV, the immutable version locks Ω_{VAR} and Ω_{REV} and a special lock to remove resources from versioning, the BL lock.

What kinds of properties these new versions have has also been described. This behavioral model includes how new versions move from the version sphere, the VSOC, to version repositories when they are made immutable, and how things work when transactions and Xymphonies commit and abort.

Toward the end of this part of the chapter, two problems were given some special attention. First, an outline of how versioning granularity can be dynamically altered by fragmenting and merging pieces of versions was given. After that, a possible combination of CCSR and MCC was described.

5.5 Versioning of configurations

Note that the ideas presented in this part of the chapter, about configurations and their use in Xymphonic versioning place some very strict demands on the requred locks to perform configuration versioning actions, and that they should be viewed as a possible starting point for further efforts.

A definition of the conceptual idea of configurations was given in section 4.2.4. As the name indicates, the purpose of configurations is to hold consistent collections of resources according to a given criterion. Configurations should also be able to hold information about a certain collection at a certain point in time, that is, of a particular version. For example, one version of a configuration could be a fix[5] of a certain bug involving the resources in a particular (version of a) configuration.

In the introduction to this chapter, a quotation from Kaiser (1995) was given as an argument in favor of configurations. Additional examples

[5] Or it could be a part of the evolution of a fix, or an enhancement, extra feature, etc.

of the usefulness of connecting a collection of resources to a common change are:

- **ASGARD** (Micallef & Clemm 1996)
  which is implemented on top of ClearCase and allows changes to different resources to be collected in an *acitvity*.
- **Stellation/Coven** (Chu-Carrol 2001)
  is an IBM research prototype which uses the notion of *consistent project versions*. To achieve this, updates to individual artifacts are recorded as an atomic change.

The difference between a configuration and an *inner* resource referencing other resources should be made clear. Basically, an inner resource references the resources it hierarchically, logically owns and belongs with. These connections usually survive through versions of the individual resources.

Versions of configurations, on the other hand, may want to reference collections of trees of resources, i.e. many inner resources, in order to logically group all the involved resources of a particular change.

5.5.1 Relations involving configurations

From the NIAM model of required relations shown in figure 5.12, we can extract the following many-to-many relations:

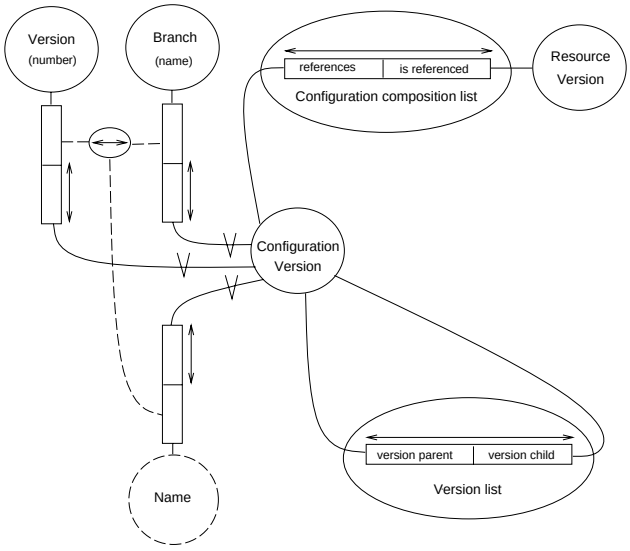

Figure 5.12: NIAM model of configuration relations.

- **The configuration composition list**
  This relation gives a list of which version(s) of which resources are referenced from which version(s) of which configurations.

- **The version list**
  This list contains the edges in the version graph of configurations.

5.5.2 Composing a configuration

In order to compose a configuration, we must first identify the resources we would like to include. Identification of resources is described in section 5.4.2. Based on an extensional list or intensional predicates or a combination of the two, we may construct a new configuration.

Note that with the scheme described here, it is required that the transaction creating the configuration version should version write lock the involved resource versions. This is required because generating versions of the configuration is dependent on all involved resources finally being commited to the controlling transaction. It may be considered natural to create configurations in TLTs. A small example of a newly constructed configuration, C referencing the resources RC_n, RCC_n, RS_n and RSS_n is shown in figure 5.13.

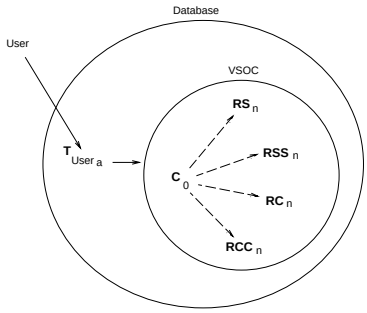

Figure 5.13: An example of a newly composed configuration.

5.5.3 Identifying versions

The process of identifying a configuration is quite similar to that of resources. That is, we can ask for a version number or create a query based on values of attributes. This query may include a choice of branch if variants exist.

In addition to those methods in common with the resources, we should

be able to identify configurations, at least partly, based on which (versions of) resources are contained and what the reationale behind creating the configuration was, e.g. we may want to find the configuration adding a feature called `zap-zap` to a laser design.

5.5.4 Selecting a version

Selecting a version of a configuration is thought to be performed in the same manner as for resources. Version write locking a version of a configuration requires acquiring version write locks on the referenced resources of the particular version of the configuration as well.

5.5.5 Creating versions, recording change

We should separate between situations we may or may not prefer to result in a new version of a configuration, and the situations which *must* result in a new version. A new version of a configuration must be created in the following situations:

1. **Applying an Ω_{VAR} or Ω_{REV} lock to the configuration**
   If the configuration is version write locked, this commits the configuration version in the local VSOC to the local repository.

2. **Transaction or Xymphony commit**
   This commits the configuration version(s) to the version repository of the enclosing environment. If we are commiting a transaction, this may either be a Xymphony, a parent transaction, or the top level DB. If we are commiting a Xymphony, it has to be a transaction.

3. **Altering the contents of the resource collection**
   Adding or removing a resource from the configuration should produce a new version.

If we apply either an Ω lock to a configuration which is version write locked in the VSOC of a transaction, this explicitly creates a new immutable version of the configuration as well as the referenced (chain of) resource versions. To be able to perform this, the transaction must have write control of the configuration as well as *all* the referenced resource versions.

Recall from section 5.2.2 that substransactions may create new versions from the contents of the repository of an ancestor transaction. This makes it possible for a subtransaction to create a variant or a version child of a configuration residing in a repository owned by a transaction

higher in the hierarchy. When creating a variant, it follows that variants of the referenced resources of the configuration must also be created, thus it is required that resources referenced from a configuration allowing variants must also allow variants. These new variants should be tagged according to the attribute/value scheme as explained for variant resources in section 5.5.3. An example of creating a configuration variant is given in figure 5.14.

When the subtransaction of figure 5.14 successfully commits to the parent transaction, the result is as shown in figure 5.15. It is implicitly assumed that the resource variant versions referenced from configuration C version 1, variant *b* also have the variant attribute *b* to separate them from those in the other branch.

Note that if a subtransaction version write locks a resource referenced from a configuration residing in a version repository, this does not neccessitate the creation of a configuration version to accompany it. If the subtransaction commits, the new resource revision will be reachable from the configuration by version graph traversal. It is an open issue how this should be handled when a new configuration is created.

If we wish to make a configgration available to other selected users through a Xymphony, we may commit the configuration and referenced resources in the VSOC to a Xymphony-owned version repository. For example, if we would like other users to contribute to the configuration in part 1 of figure 5.16, the situation would be as shown in figure 5.17.

An example of adding or removing a resource from a configuration is shown in figure 5.16. The initial state is shown in part 1 of figure 5.16. When we remove a resource, as in situation 2 of the same figure, we must record the state of the configuration before this addition in order to later track down this transition. This means recording the current versions of the resources as well as the configuration to the version repository. The case of adding a resource as shown in part 3 of figure 5.16 is based on the same principle.

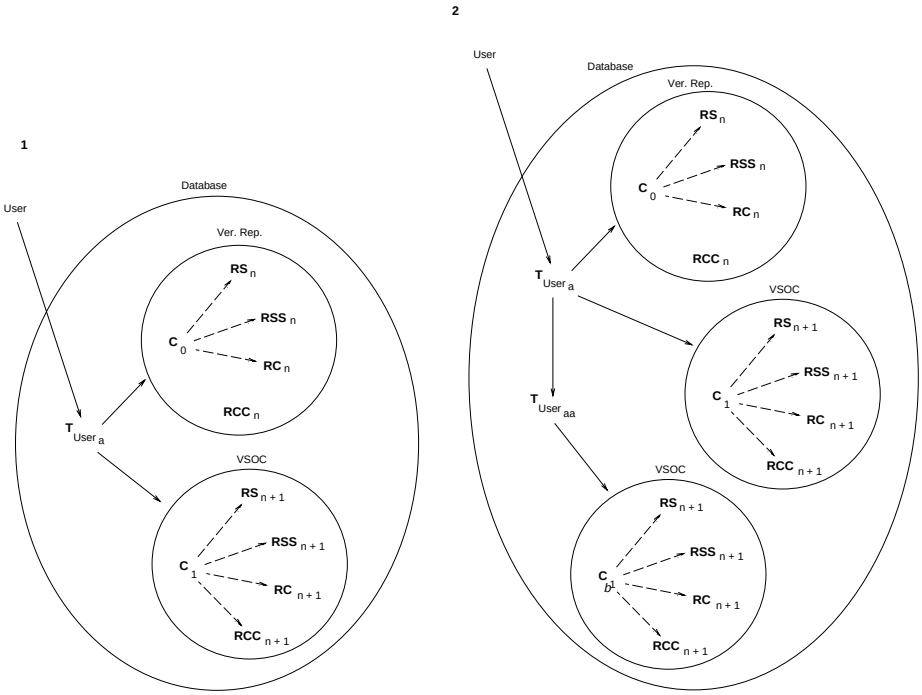

Figure 5.14: Creating a variant of a configuration and working on it in a subtransaction.

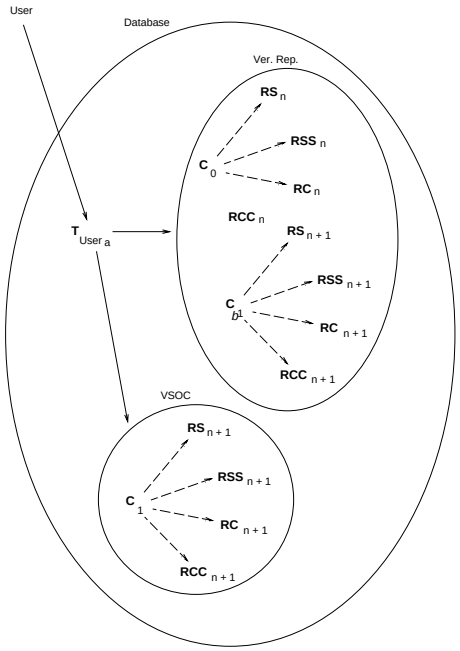

Figure 5.15: The situation after the subtransaction in figure 5.14 has commited.

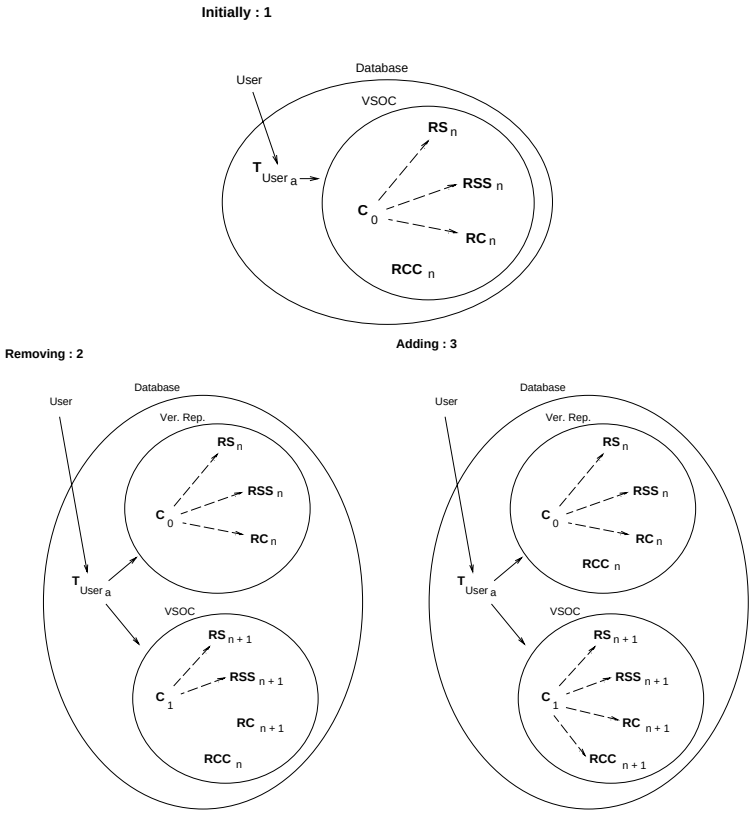

Figure 5.16: Adding or removing a resource from a configuration.

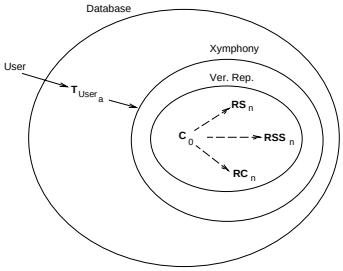

Figure 5.17: Delegating a configuration to a Xymphony.

5.5.6 Performing merges

In order to merge configurations the following questions should be answered:

- If the configurations reference different versions of some of the same resources, should the newer replace the older, should they also be merged in some way, or should they coexist in some form?
- If (some of) the configurations stem from a common ancestor, what consequences should this have?
- What consequences should it have if they do not stem from a common ancestor?

It is probably fair to assume that the transaction responsible for the merge must either have version write locks on all involved resources and configurations, or the idea with read-to-merge locking as described in section 5.4.5 should be employed in some form.

5.5.7 An idea of a typical work pattern

We have seen that the creation of a new version of a configuration places quite drastic demands on lock requrements. However, creating a new version of a referenced resource should not neccessarily lead to the creation of a new configuration. Based on this, a sensible work pattern could, for example, be the following:

1. Create a new configuration, responding to a particular task, in the TLT.
2. Delegate the resource versions through xymphonies/subtransactions.
3. When the TLT regains write control of all the referenced versions, including new resource versions produced, we may conclude that this would equal an iteration in development. Accordingly, the TLT owner may choose whether to commit this increment, and thus create a new version of the configuration. Or she may Ω lock the configuration version (and the resources with it), and start on the next iteration by once more delegating the (latest) resource versions.

5.5.8 Configuration versioning summary

As argued both in the introduction of this chapter and in the beginning of section 5.5, configurations are an indispensable ingredient in any serious version management system. Currently, to create a new version of

a configuration, the acting transaction must have a version write lock on both the configuration version and the resource versions referenced from the configuration, or employ a special read lock trick.

This may be considered an unduly strict demand, but it should be pointed out that creating a new version of a referenced resource does not neccessitate the creation of a new configuration version. An idea of a typical work pattern was presented in the preceeding section.

5.6 Relation to other models

The task of comparing the ideas behind and consequences of the model presented in this chapter with other models is not as easy as one might think. This is due to two facts:

1. Not many other models for long lasting transactions incorporate versioning concepts.
2. Not many other versioning models incorporate concepts from transaction management, esp. long lasting transactions.

To accentuate these two current facts, a quotation from Conradi and Westfectel (1998) helps:

> It has been recognized for a long time that the ACID principle cannot be transferred from short to long transactions [Barghouti and Kaiser 1991; Kaiser 1995; Feiler 1991a]. Rather, precommit cooperation is required in order to coordinate long-lasting development and maintenance tasks. Customizable policies have been developed to control cooperation. Many approaches to long transactions do not take versioning into account [Barghouti and Kaiser 1991]. This is a severe restriction since versions play a crucial role in cooperation control [Estublier and Casallas 1995]. So far, only a few SCM systems support long transactions [Conradi and Malm 1991; Godart et al. 1995]. Many others merely provide workspaces and mechanisms for controlling change propagation between them [Estublier 1996].

Transaction models for long lasting transactions seldomly incorporate versioning concepts and when they do, it is as far as I have been able to ascertain, in order to help concurrency. In our case, concurrency is already handled by Xymphonic transactions. Ergo, there is no appropriate taxonomy available for giving a true comparison between the model presented in this thesis and other transaction models. Note however

that in (Anfindsen 1997, 93-102), the Xymphonic model itself is compared to related models and techniques.

It should also be noted that version management tools which claim to support transactions mostly do so by crude locking and isolation of workspaces of individual participants. Variants may in some cases be created and later merged into the main line of development. As pointed out by Zeller (1996) this technique is mostly inefficient and has several disadvantages. It seems promising that a Xymphonic model for version management could be a valuable contribution in this respect.

5.7 Chapter summary and conclusions

This chapter has described of a possible combination of the Xymphonic transaction model with versioning concepts. It is based on some new lockmodes, a new SOC and a storage container called the version repository. The new lockmodes serve the following purposes:

- To give compatibility answers.
- To denote if a version is immutable. If so, it is locked with one of the Ω locks. If it allows only one version child, a single revision, it is locked with the Ω_{REV} lock. On the other hand, if it allows potentially any number of parallel version children, variants, it should be locked with the Ω_{VAR} lock.
- To denote that we are writing on a new version. If this new version is going to allow only one version child unless explicitly Ω_{REV} locked sometime, it should be REV locked. Conversely, if we want to allow variants we should use the VAR lock or else later use the Ω_{VAR} lock explicitly.
- To remove a resource from versioning we would try to BL lock it.

When a resource is version write locked by either the VAR or REV lock, it is located in the new SOC, called the version SOC (VSOC), of the owning transaction. When a versioned resource is commited, it is made immutable and Ω_{VAR} or Ω_{REV} locked to be stored in a version repository. This version commit can be performed either explicitly by manually setting the Ω lock or implicitly by commiting the transaction.

Along the way, solutions to the problems of dynamic version fragmentation and the issue of combining CCSR and MCC were outlined.

To summarize, the features offered by the enhanced model include:

- Well known Xymphonic functionality such as:
  - Long lasting collaborative Xymphonic transactions.
  - Delegation of work through Xymphonies.
  - Communication of state and relaxation of the ACID serialization properties through parametrized locks.
- New versioning functionality such as:
  - Creation of versions of data items, resources.
  - Choice of variant or revision semantics.
  - Variants through branch identity.
  - Representation of version histories through version graphs.
  - Outlined support for configurations.

Chapter 6

Further work

This chapter presents some possible future lines of research based on the results presented in the thesis. It also contains some questions which for some reason, typically lack of time, were never investigated to the degree they deserved but nevertheless, would be interesting to pursue.

6.1 Version locks and concurrency

As pointed out in section 6.2, chapter 5 does not discuss when it is preferrable to use variant or revision semantics. An identified problem is that sometimes, a user may want to ensure that branches are not allowed, thus prohibiting people from working on parallel versions in different transactions. Note that concurrency is still assumed to be handled by the Xymphonic model, and variants are allowed to provide, for example, temporary fixes, special modifications and parallel exploration, as explained in section 4.2.3 and versatility as described in section 4.3.

The problem of using variants as a vessel for undesirable concurrency can be partly avoided by making sure the immediate version parent of the object locked by the transaction is Ω_{REV} locked in the pertinent Xymphony, ancestor transaction, or the top-level DB. However, this solution does not preclude transactions from creating variants of earlier versions if they have been commited with variant semantics.

One possible solution could be to introduce a view containing information about active transactions in the process of creating versions, their policies regarding variant cooperation from other transactions, possibly in addition to a larger rule-base of allowable variant creation. For another transaction to create a variant it would, for example, have to

satisfy both the requirements set forth by the active transaction as well of those of a more static character in the rule-base. A solution in this direction would make it natural to reduce the VAR and REV lockmodes to only one version lock, and the Ω_{VAR} and Ω_{REV} to one Ω lock, and instead control their semantics by for example the means proposed in this section.

6.2 Change of versioning semantics

As it is pointed out on page 53, the consequences of the changing between the different combinations of versioning semantics is yet undefined. For example, attempting to REV lock a resource version with variant semantics signals a desire to restrict possibilities. This may be fine. But what should happen later if it is discovered that it is imperative that this particular version produces variants? One possible solution may be to allow attributes of versions to be altered by placing an X lock on it. This should, no doubt, be restricted by some criterion.

Although we may run into trouble by restricting the semantics to revision-only, it seems probable that the problems could be worse if a version with revision semantics is altered to allow variants. One problem is that allowing variants may not be what was initially thought desirable. However, after changing to variants and creating a possibly large branch leads to a problem when someone suddenly realizes that it , for example, goes against some business rule. It seems highly undesirable to delete the whole tree and thus discard the work done. On the other hand, merging the variants with the main line of work may not either be easy, perhaps not even possible.

Based on the arguments made in these two paragraphs, it would seem advisable to adhere some restrictions on the change of versioning semantics. If the Xymphonic versioning model were ever to be used in practice, this should be investigated further, but a fair guess is that making the restrictions dependent on e.g. project strategies and goals are sound requirements. Also note that if the ideas in the preceeding section were to be followed up, that is, the VAR and REV locks being replaced by a single version lock and the two Ω locks being reduced to one, the concerns pointed out in this section would probably be solved along the way.

6.3 Further work on configurations

As stated in the preceeding chapter, work remains to be done on configurations. This includes exploring the following:

- How to loosen up some of the most rigid requirements. Alternatively, perhaps, finding that they are indeed neccessary.
- A discussion of possible approaches to and outcomes of merging configurations.

6.4 Intent locks and configurations

Intent locks, also known as granular locks, are a means to increase concurrency by weighing lock management overhead against concurrency, and they also remove the serializability failure of phantoms (Gray & Reuter 1993, 406). The intent locks introduced in section 3.3 are the following:

1. **IR** - Intent to read.
2. **IW** - Intent to write.
3. **RIW** - Read with intent to write.

It would seem worthwile to investigate the pros and cons and possible efficiency gains of using intent locks with configurations of resources. It may become even more important if we were to allow for configurations to further reference configurations in a recursive manner.

6.5 User interaction patterns

A common demand (esp. in open-source development) on SCM systems is the ability to approve changes before they are commited. There are many solutions to this, but none of them is as elegant as the Xymphonic, where one simply commits the changes back to the supervisor which may or may not approve, or may chose to delegate the approval to a board of peers.

Presently a standard technique is to send commits in a human-readable diff form (compared to the parent revision) by email(s) to mailinglists[1], or posts to newsgroups where they are reviewed/corrected by a committee and eventually commited or discarded. This is a common way of

[1] One example is the Subversion (`http://subversion.tigris.org/`) project.

discriminating between the different trust levels of developers. Those less trusted must commit through email and be reviewed while the ones with proven skills have access to commit new versions directly to the repository.

While these procedures do not have the elegance and functionality of Xymphonies and NCCSR, they are much simpler both to understand and implement, and everyone with a mail client can potentially participate. However, if a system with Xymphonic functionality is properly implemented, it is fair to assume that the complexity of use would favor the Xymphonic approach.

It seems a worthwile endeavor to investigate the applicability of distributed Xymphonic transactions with versioning in this kind of environment, with many participants who are loosely connected and have different trust levels.

6.6 Process support, active databases

It would seem that without altering the transaction model, the responsibility of using one process model or another lies on the project participants. There is no special support described in this thesis, but it is conceivable that it could either be added as a separate layer in an implementation, or as an addition to the model itself.

This model enhancement could, for example, be based on configurations. An example is a version of a configuration being equal to one rotation in a cycle of iterative, incremental development. It would be interesting to see what kind of opportunities lie in this direction.

Another approach could be based on active database systems technology. As stated in the chapter on further work in (Anfindsen 1997), active database systems could provide interesting research in connection with Xymphonic transactions. Anfindsen gives examples of areas where it seems probable that the Xymphonic model is in a position to enrich the field of active databases, particularly with visibilities between transactions of interacting heterogenous components, and also the problem of rule execution and serializability.

In the name of process support, it would be interesting to turn things the other way around and see in what ways ideas of active database systems could be used to enrich Xymphonic collaboration, especially with versioning in mind. For example, if a transaction has created some new

versions of an object and commits its results, it is conceivable that we may want these new versions to trigger actions in a process support scheme. It has to be further explored whether or not this is a usable idea.

6.7 Change oriented features, deductive databases

Conradi and Westfechtel (1998) write the following:

> Deductive databases [Das 1992; Ramamohanarao and Harland 1994; Ramakrishnan and Ullman 1995] provide for persistent storage of facts and rules and are usually based on a Prolog-like data model. Deductive capabilities are urgently needed for intensional versioning. On the other hand, deductive databases have been employed only rarely in SCM [Zeller 1995; Bernard et al. 1987; Lavency and Vanhoedenaghe 1988]. Rather, many SCM systems incorporate homegrown deductive components that have been developed in an ad hoc manner.

It would be interesting to see if change oriented versioning features could be integrated into the Xymphonic model for versioning, and work together with the ideas of this thesis. In connection with an effort in this direction, it would be natural to see in what way deductive database systems could be used to provide the DB functionality.

6.8 Temporal databases

Temporal database systems are entirely focused on the time dimension, they do not, for example, cover variants. However, the Adele SCM system (Estublier & Casallas 1994) uses temporal database functionality to store revisions, and other mechanisms to handle e.g. variants. It would be interesting to study the degree of usefulness of temporal databases for Xymphonic transactions and versioning.

6.9 Distributed databases

The Xymphonic model in connection with distributed databases is the topic of (Anfindsen 1997, 85-92). If pursuing the possibilities outlined in (ibid), it would be natural to also investigate whether the ideas of this thesis have any bearing, positive or negative, on the usefulness of Xymphonic transactions in a distributed setting.

6.10 Further implementation

Although this thesis presents a prototype implementation of the most important versioning ideas presented, interesting implementation questions remain to be answered. The prototype presented in appendix A does not concern itself with parameter sets, neither is it intended to be optimal with respect to speed and storage efficiency. Also, configurations are only implemented to a very limited degree (they are not even visible to the end-user).

This raises at least three new questions:

- Can configurations be implemented in an efficient and useful manner?
- How can parameter treatment be implemented alongside versioning, what practical program design questions does this raise, and how can it be solved?
- What is the best way to go about optimizing an implementation of Xymphonic versioning for speed and storage efficiency? Can any new algorithms or datastructures be discovered which are of particular use for this model?

Finally, it would be appealing to investigate the possibilities of implementing a virtual file system with Xymphonic versioning functionality. This has been done for the ClearCase and ICE SCM tools, but can it be practically implemented with Xymphonic functionality?

6.11 User interfaces

What are the best ways of presenting Xymphonic versioning functionality to an end user? In the case of creating a serious implementation, it is important to have qualified ideas about how they will react to different forms of interfaces.

6.12 Chapter summary and conclusions

This chapter has presented some new possibilities which have arisen in the course of finding a suitable way of adding versioning features to the Xymphonic model. It also includes items which due to time limitations were never explored and solved to a satisfactory degree.

Chapter 7

Thesis summary and conclusions

This thesis had as a main objective to discover a solution of combining versioning concepts with the Xymphonic transaction model in such a way that it was beneficial for either one or both of the fields involved. To provide background for these enhancements, chapters on transactions and database concepts as well as concepts of version management and an overview of the Xymphonic model were presented in the beginning of the thesis.

In order to motivate a combination of the Xymphonic model with versioning, it was pointed out that there are valuable properties to be gained which were not a already a part of the transaction model. When allowing revisions to be created in a controlled manner, the following benefits are gained:

- **Backtracking** when we need to go back to an earlier version
- **Exploration** of the evolution of an item
- **Exploratory development** is possible when we know we can revert to a safe state
- **Comparison** is possible if we want to see what has changed from one version to the next
- **Safety** in the sense that we can go back to a particular version and fix errors there when they are discovered
- **Safety** in the sense that we have versions known to be stable when a system is undergoing big changes
- **Rationale capture** by writing comments for each new version

- **Reuse** by being able to use suitable versions of objects in other projects

At least to a certain degree, these advantages would also be availiable by tailoring the Xymphonic model for use with a temporal database system. However, it is a great asset to be able to allow for version branches, variants, of objects to be created. This is not at the present time supported by temporal databases. Allowing variants gives the following benefits:

- **Temporary fixes** to older revisions
- **Special modifications** which are not natural in the main line
- **Parallel exploration** when we do not know which approach is best
- **Versatility** when we need different variants for language, operating system, etc.

The problem definition of the thesis, divided the main problem into three adherent problems:

1. How, if at all, can versions be generated in a controlled fashion in a tree of transactions and Xymphonies?
2. How, if at all, can configuration control be incorporated into this scheme?
3. Is it possible to create a model which is also feasible to implement?

To provide for a natural progression towards an answer to the main problem, the adherent questions will be answered one by one.

7.1 Creating versions

To allow for revisions and variants to be created, two new lockmodes were introduced: The VAR lock for creating variants, and the REV lock for creating revisions. Also, when a version of a resource is commited, either within its transaction or as a consequence when the owning transaction commits, the version is made immutable. This is marked by the Ω lock. When it is not implicit whether an immutable version allows only one child version (revision) or more than one (variants), the Ω lock is subscripted with $_{REV}$ or $_{VAR}$ respectively. Also outlined is how a resource can be removed from versioning by locking it in exclusive mode and then applying a blank (BL) lock.

When a versioned resource is locked for writing by a transaction, it belongs to the version SOC (VSOC) of the transaction. The VSOC is a means of explicitly separating the write locked versioned items from those that are unversioned in order to increase clarity in explanations. The write locked unversioned items will belong to the WSOC of the transaction as before.

If we would like to create more than one version of an item within the lifetime of a transaction, we can Ω lock the item explicitly, as opposed to implicitly Ω locking it on transaction commit. This will make the version immutable, move it to the version repository of the transaction, and return a version lock on a new child version. The version repository is a store of immutable, Ω locked objects.

It is also possible for a group of collaborators to cooperate on creating new versions through Xymphonies. Each Xymphony also has a version repository. A user may chose to make an item he has in his VSOC available to other users by Ω locking it to the version repository of the Xymphony. Other users may VAR and REV lock version children of the item in the domain of the Xymphony.

It must be concluded that, yes, versions can be generated in a controlled fashion in a tree of transactions and Xymphonies.

7.2 Configuration control

As pointed out by Barghouti and Kaiser (1991) and later by Kaiser (1995), an important part of versioning systems should be configuration control. That is, a means of marking which resources are consistent with each other in some respect. This can for example be to enable software developers to tell which parts of a program were included in adding a particular feature. Some support for configurations is included, but so far only a minor part of the solution for version extending Xymphonic transactions. Regrettably, configurations were easily integrated only with special, restrictive considerations being made regarding locking. As reflected in the chapter on further work, configurations require further efforts to be an effective tool for the Xymphonic versioning model.

It can be claimed that a partial solution has been found for the problem of configurations. Some further lines of research were identified in this respect.

VAR, REV	Version write locks
Ω_{VAR}, Ω_{REV}	Version locks for immutable versions
BL	Lock which removes a resource from versioning
Configurations	Maintaining consistent collections of resources to suit special purposes
VSOC	Sphere of control for VAR and REV locked items
Version repository	Store for immutable versions located in transactions, Xymphonies and top level DB

Figure 7.1: Main contributions of the thesis

7.3 Implementability

The theoretical work culminated in a prototype implementation of the most central concepts, namely the VSOC, the version repository, the VAR and Ω_{VAR} locks. Even though configurations were implemented only to a limited degree, this serves as a proof that the ideas presented were of a practical nature and also feasible to implement. A high-level description of the implementation is given in appendix A.

7.4 Main problem

Based on the results obtained by theoretical experimentation, presented in chapter 5, and the prototype implementation presented in appendix A, it should be fair to claim that versioning and Xymphonic collaboration form a powerful symbiosis.

Of the three adherent questions to the main problem, two out of three were given a solution. Unfortunately, configurations were not as easily integrated as one might have hoped, but if further time and research is vested on the subject, elegant solutions might be produced.

The main theoretical results of the thesis are shown in figure 7.1.

Appendix A

Proof-of-concept implementation

The technical content of this chapter is restricted due to confidentiality agreements signed by Xymphonic Systems AS and me on February 16, 2001, and March 4, 2002.

A.1 Overview

It is a well known fact that theory becomes increasingly interesting and relevant when it is backed up by concrete experiments and data. In order to provide this, a subset of the functionality outlined in chapter 5 was implemented in a proof-of-concept program. The intention of the implementation was to create some functionality of a distributed, Xymphonically transaction managed filesystem with versioning capabilities. The implementation does not claim to be bug-free.

The implementation consists of about 11300 lines of code divided among three major parts: Client($\approx$ 4400 lines), server($\approx$ 3100 lines) and lock manager($\approx$ 3800 lines). The lock manager with versioning capabilities was based on an exisiting implementation of Xymphonic locking by Anfindsen which was about 2500 lines. Admittedly, these figures would be different if some of the code had been refactored, particularly in the client, and functionality had been relocated, particularly from lock manager to server.

The major parts of the implmentation are organized in the following packages:

- **Client**
  `com.xymphonic.clientfrontend`

- **Server**
  `com.xymphonic.versionserver`
  `com.xymphonic.persistentdata`
  Where `versionserver` contains the server functionality and `persistentdata` contains the data classes to store in Objectivity.
- **Lock manager**
  `com.xymphonic.lockmanager`

A.1.1 Programming language

Based on previous positive experiences, Java was chosen as programming language. These reasons include the following:

- Nice syntax
- Powerful / expressive enough
- Automatic garbage collection
- Platform independence
- Good supply of bindings to DBMS's
- Ok API for programming GUI with the swing components

A.1.2 Database backend

Objectivity, an OODBMS, was used to provide the base DB functionality. This choice was also based on previous good experiences. Important reasons for choosing Objectivity were:

- Flexible, object oriented storage of data.
- Query facilities, including both SQL and an Objectivity proprietary language on object attributes.
- Good DB browsing and query capabilities with an Objectivity program called the ooAssistant (using an Apache HTTP server). This is convenient when debugging.
- It has a good language binding with Java, and this gives us the following benefits:
  - We can marshal database objects over the network to the client through Java RMI.
  - Integration with server code is seamless and thus reduces complexity and improves programmer productivity.

A.2 Architecture

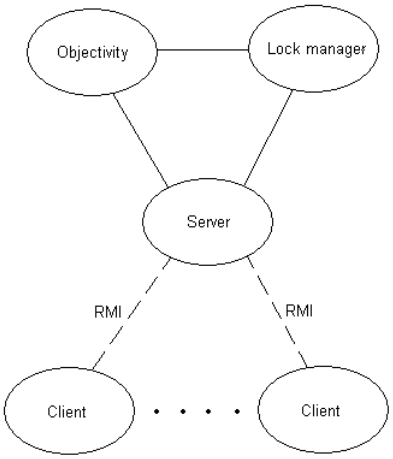

Figure A.1: The implementation architecture

All parts are implemented using Java. The lock manager is built on a standard Xymphonic lock manager, which is extended with version locking. It has ties to Objectivity because when a version lock is granted, a new persistent object is created. The server receives remote method invocations from the client and acts on these. This can, for example, be either to query Objectivity in order to update some client side view, or it can be operations on the lockserver. The implementation is based on file data, which is imported into Objectivity. This is also done on the server side. The client side provides a front end for giving commands and editing and viewing files and lock manager status.

A.3 Client

The client module, `com.xymphonic.clientfrontend`, does not contain any particularly exciting implementation details. It provides a graphical interface to the functionality provided by the server and the lock manager. An explanation of the structure and functionality as well as some screenshots will be given.

A.3.1 Structure

This section lists the central classes of the client and explains their purpose.

- `ClientGUI`
  This is the main class of the client.

- `EditorDialog`
  It is used to edit file contents. To do this, a standard JTextPane is used. If the user is holding a read lock it may be used to view the contents of the file.

- `ResourceHistoryPanel`
  This class subclasses JPanel and is used to show the version history of a resource. It also contains buttons to lock a version of the resource or create a new variant.

- The client package also contains some subclasses of JDialog in order to ask the user about different data as well as listener classes.

A.3.2 Functionality

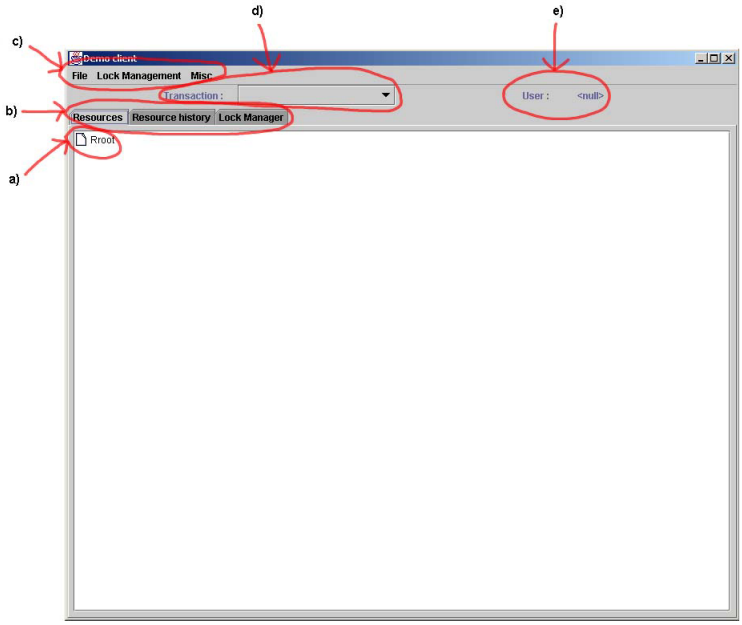

Figure A.2: The client GUI at startup. Explanations are given for the encircled parts.

a Display of resources (files and directories) and the contents of the lock manager is done with JTrees. The root node of a JTree can

not be altered at runtime, so a static root called Rtree is used in the Resources pane.

b A JTabbedPane is used to switch between the view of resources (figure A.3) , resource history (figure A.5) and lock manager state (figure A.4).

c A JMenuBar holding the command options.

d A user may have more than one transaction in progress. This JComboBox is used to choose in which one a chosen command should operate.

e The name of the logged in user, <null> if not logged in.

The command options offered are the following (for each choice on the menubar):

- **File**
  - **Login** - Locates and opens a connection to the RMI server.
  - **Resource editor** - Opens up an editor if we have selected a resource with the proper lock in the lock manager tab in the tabbed pane.
  - **Exit** - Exit the program.
- **Lock Management**
  - **Begin Transaction**
  - **Commit Transaction**
  - **Abort Transaction**
  - **Begin Xymphony**
  - **Commit Xymphony**
  - **Abort Xymphony**
  - **Lock Resource**
- **Misc**
  - **Reload lock manager tree**

In addition to the commands listed here, the Resource History pane has a button to create and lock a variant of a resource, if possible.

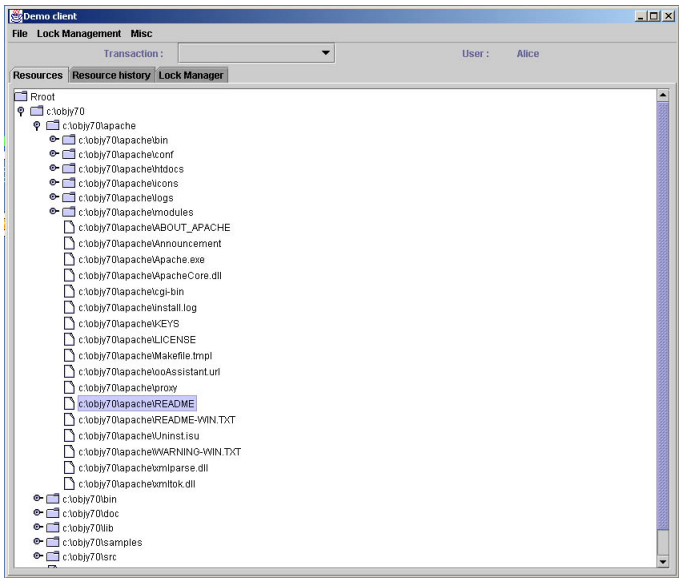

Figure A.3: The resources in the DB. In this case it contains the Objectivity 7.0 trial version for Java. The file README is selected. Note that c:\objy70\apache\README refers to the location the file was imported from, not where it now resides.

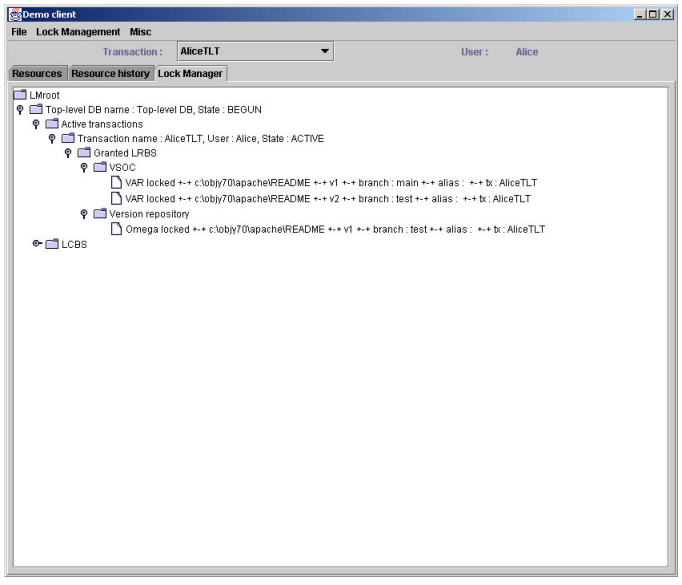

Figure A.4: The user, Alice, has started a top level transaction called AliceTLT and locked the resource from figure A.3 with VAR and Omega locks and also created a branch called test.

A.4 Server

The server module, `com.xymphonic.versionserver`, is responsible for interacting with the lock manager, querying Objectivity and creating datastructures suitable for display on the client, and creating and inserting initial resources into the DB. This functionality is exported to clients through a subclass of the `UnicastRemoteObject` implementing a remote interface.

A.4.1 Structure

The most important classes of the server are the following two:

- `Init`
  This class is used to create storage containers used by this implementation in a blank Objectivity DB.

- `MainClass`
  This class is responsible for receiving method calls from clients, delegating and replying to these in a proper format.

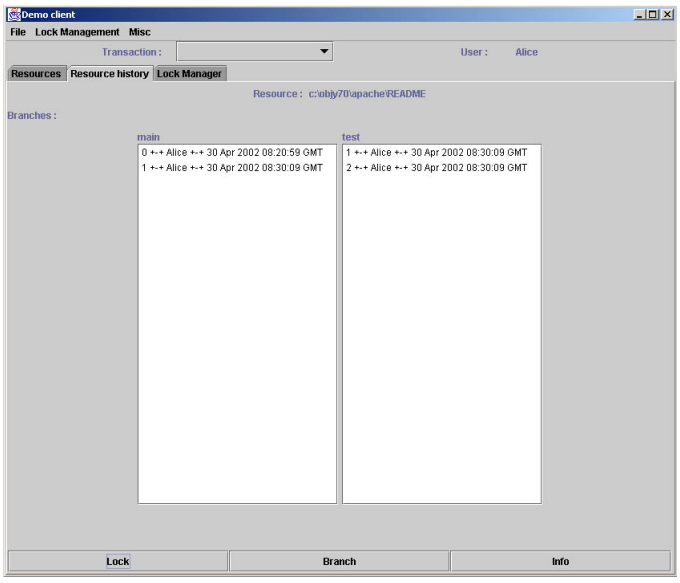

Figure A.5: The version history of c:\objy70\apache\README after AliceTLT has been commited.

A.5 Persistent classes

These classes are located in the `com.xymphonic.persistentdata` package. In order for a class to be accepted as persistent by Objectivity, it must either subclass `com.objy.db.app.ooObj` or if it is already subclassing another class, or if there are other reasons, it must implement one of the persistent enabling interfaces defined by Objectivity. Note that most standard Java classes and types such as `String, int and Date` are automatically made persistent if referenced from a persistent objectin Objectivity. If a value should not be persistent, the keyword `transient` must be used in the variable declaration.

Some code listing will be given in later sections. The persistent classes of this implementation include the following core classes:

- `Resource`
  This class holds the contents and identification of a version of a resource.
- `Branch`
  Contains the name of the branch of a version.
- `VersionNumber`
  Contains the name of the branch of a version.
- `Rid`
  Contains the resource id, e.g. c:\objy70\apache\README. Together with the version number and branch, this identifies a version of a resource.
- `Contents`
  Contains the byte contents of a file. For a directory it will be null.
- `Configuration`
  References a number of versions of resources to serve a particular purpose. In this implementation, the `Configuration` class is used only for easy access to stored resources by the server classes. Users do not have the opportunity to create and act on configurations.

In the design phase, it was decided to keep the names and class interfacing as generic as possible for possible later reuse with items other than straight files and dirs. For example, the `Resource` class is not called `PersistentFile` and it was also chosen not to integrate the branch, version number and resource id directly into the `Resource` class in the form of `String / int` values. This fits well with the software design

paradigm of high cohesion/low coupling. That is, conceptually the objects are tailored to fill well defined roles (high cohesion) and at the same time, they provide accessor methods to separate communication from concrete variables and storage (low coupling).

A.5.1 The Resource class

Note that this codelisting only shows the relevant fields of the class. The methods, constructors and variables used for temporary book-keeping are omitted. A few notes are given for some of the fields.

```
public class Resource extends ooObj {
  // persistent fields :
  protected Contents contents;
  protected Rid RID;
  protected Resource[ ] versionAncestors;
  protected Resource[ ] versionSuccessors;
  protected Resource[ ] resourceParents;
  protected Resource[ ] resourceChildren;
  protected int type; // 0 = inner, 1 = outer                          10
  protected LockParameters lockParameters;
  protected VersionNumber versionNumber;
  protected Date timeStamp;
  protected Alias alias;
  protected Branch branch;
  protected User responsibleUser;
  protected String versionDescription;

  // Type of omega lock :
  // 0 = unversioned, 1 = revisions only, 2 = revisions and variants   20
  protected int semantics;

  // legal values for semantics field
  public static final int UNVERSIONED = 0;
  public static final int REVISION = 1;
  public static final int VARIANT = 2;

  // legal values for type field
  public static final int INNER = 0;
  public static final int OUTER = 1;                                   30

}
```

Resource references

These are the `resourceParents[]` and `resourceChildren[]` arrays. They are used to denote the hierarchical relationship between resources. To

provide generality, resources can have multiple resource parents and children. Because this implementation is based on standard files and directories, the `resourceParents` array will point to the parent directory and thus never use more than the first index. If it is a file, all indexes in `resourceChildren` will be null.

Objectivity has functionality for describing and managing relations between objects. It would have been appropriate to use a `ManyToMany` relationship in this case. But in order to get the implementation up and running quickly, with a low complexity, a simple array strategy was chosen.

Version references

These are the `versionAncestors[]` and `versionSuccessors[]` arrays. The implementation does not support merging of versions, so at most the first index of `versionAncestors` will be used. This method of representing a graph, i.e. by having a list of edges at each node, is known as the adjacency list datastructure, and it is the preferrable structure for a sparse graph[1].

Finding versions is not done by traversing the graph, but by using the query facilities of Objectivity.

A.5.2 Storage strategy

Ojects in the implementation are stored uncompressed in an Objectivity DB. That is, each resource version is stored in full without any compression in terms of common parts with versioning relatives. However, there are possibilities of later enhancing the program with deltified storage. Note that the following discussion assumes directed deltas. The `Contents` class has the following fields:

```
protected byte[] contents;
protected boolean deltified;
protected DeltaModelInterface deltaModel;
```

Disk usage

It is thought that if the `deltified` variable has the value `true`, the bytes of the `contents` field are compressed in terms of other versions' contents using methods in a delta model satisfying the requirements of a

[1] In most cases we can not assume that a version will have more than one or two children and one or two parents

`DeltaModelInterface`. The uncompressed form can then be constructed by calling methods of the same object, passing contents of suitable versions as parameters.

Also, it is possible to compress the contents in terms of itself, for example with the utilities in the `java.util.zip` package. An interesting note is that it was shown in the Vdelta technique (Korn & Vo 1995) that compression is a special case of deltification. This means it is feasible to implement a general diffencing and compression engine which could then be used to provide very good flexibility in terms of delta computation. The resultant data can, for example, be saved in the vcdiff compression format (Korn & Vo 2000). A brief explanation of deltification is given in section 4.2.7.

Version reconstruction

Reconstruction of older versions is a tradeoff between disk space usage and speed. The following are examples of strategies:

- Store a delta between each revision and leave it at that. If we leave it at this, the time needed to reconstruct old revisions can become considerable.
- Storing every nth version without compression, allowing us to start reconstructing closer to the revision in question. This gives us a much shorter chain of computations but takes more space.
- The skiplist (Pugh 1989) is a datastructure with probabilistic performance depending on a good randomizer function. It offers good space-time tradeoff by randomizing how many deltas each version should have. The higher number of deltas, the further back we can jump just by using one delta. For example, in figure A.6 we only have to apply three deltas to get from revision eight back to number one.

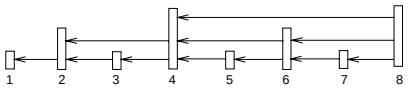

Figure A.6: An example of a skiplist.

A.6 Lock manager

The lock manager code is located in the `com.xymphonic.lockmanager` module. Due to the confidentiality agreements referred to in the beginning of this chapter, the descriptions given here will be of a limited character. The versioning lock manager was based on an existing implementation with Xymphonic functionality. Before any changes were performed, the lock manager had the following functionality:

- Transaction begin/commit/abort
- Lock/unlock items with most of the traditional lockmodes explained in (Anfindsen 1997, 20), including Xymphony locking.
- Xymphony begin/commit/abort

The following features were added:

- Tying control objects of the lock manager to implementation objects stored in the Objectivity DB.
- Adding VAR, REV, and Omega locks with their adherent compatibilities and behavior. The REV lock is not available in the client implementation, and neither is unversioned writing.
- Removing a resource version from versioning by using a BL lock was added, but has not been thoroughly tested.
- Creating and storing versions.
- Version repositories.

It should be noted that Objectivity has a lock manager of its own. This resulted in the implementation having to begin and commit/abort Objectivity sessions, roughly equal to transactions, when accessing the contents of the database. These sessions either aborted, if a fatal problem was encountered, or commited, when the server method returned (or sooner).

Due to this, it was attempted to store the whole Xymphonic lock manager in Objectivity to provide easy, serializable method invocation and persistence on the lock manager and its data. Unfortunately, this was impossible due to the massive use of a third party datastructure implementation in the old lock manager, which was incompatible with some of the rules for object storage in Objectivity. Replacing this datastructure with a compatible one would in effect mean writing a new lock manager from scratch, and this was reasoned as not being worth the

extra work.

The patchwork of synchronization resulted in Objectivity regulating the short term database access, Java synchronized blocks handled concurrency in the lock manager, while the version extended Xymphonic lock manager was responsible for the long term locking and collaboration.

A.7 Chapter summary and conclusions

This chapter has presented a proof-of-concept implementation of the central concepts of the version extended Xymphonic model. These concepts include version locking with the VAR lock, and explicit and implicit immutable locking with the Omega lock. In addition to these locks, nested version repositories were implemented and used for transactions and Xymphonies.

Based on the experiences gained by programming the prototype as presented in this chapter, it is fair to claim that version extensions to Xymphonic transactions are well suited for implementation.

Appendix B

Formal notation

When creating example scenarios and explanations, it was a frustrating experience to spend so much time and effort in order to make decent figures and subsequent verbal explanations. It was also suspected that there was a lack of clarity which was hard to overcome using the classical figure/explanation approach. This incited a testing of alternatives for explaining structure and dependencies. The result was the development of an EBNF[1] grammar and a notation for versioned items.

Unfortunately this compactness severely degraded the readability of chapters which were supposed to explain, clarify and motivate central ideas and solutions. Anyhow, because someone might some day need this notation as a starting point from where to perform more formal experiments on the version extended Xymphonic model, or may find other uses for it, it is included in this appendix.

B.1 EBNF rules for nesting

The following standard EBNF syntax is used:

|	Separates alternatives.
?	A (meta)symbol or grouping can occur 0 or 1 times.
∗	0 or more times.
+	1 or more times.
⟦ . . . ⟧	Grouping of symbols.

[1] Extended Backus-Naur form

The grammar:

< resource >	::=	**C** | **R**[‡]
< resourcelist >	::=	**Rlst** { ⟦ < resource > ⟦ , **C** | , **R** ⟧* ⟧? }
< xym >	::=	**XYM** { < resourcelist > , < vrep > , < txs >[β] }
< xymlist >	::=	**Xlst** { ⟦ < xym > ⟦ , < xym > ⟧* ⟧? }
< wsoc >	::=	**WSOC** < resourcelist >
< rsoc >	::=	**RSOC** < resourcelist >
< vsoc >	::=	**VSOC** < resourcelist >
< vrep >	::=	**VREP** < resourcelist >
< tx >	::=	**T**[†] { < txlist >[α] , < xymlist > , < wsoc >? , < rsoc >? , < vsoc >? , < vrep >? }
< txlist >	::=	**Tlst** { ⟦ < tx > ⟦ , < tx > ⟧* ⟧? }
< userview >	::=	**U**[§] { < txlist > }

‡ The **C** and **R** denote configuration and resource respectively, and should follow the rules of notation suggested in the next section.

† This **T** should be on the form $\mathbf{T_{n\,U_m}}$ where *n* is the transaction identifier and U_m is a user with identity $_m$.

§ Should be on the form $\mathbf{U_m}$ where $_m$ is the identity of the user.

α These are the subtransactions of the tx.

β These are the active transactions in the (sub)Xymphony.

An example of nesting

This textual syntax representation is equivalent to situation 3 in figure 5.11 where the generic "User" has been replaced with Alice. Note also that the graphical figure was produced with simplicity in mind and therefore, some information was omitted. The notation of the resource versions, RS_0 and RS_1, is thus not completely compliant with the proposed notation in the next section.

```
U Alice
{
    Tlst
    {
        T a U Alice
        {
            Tlst
            {
                T aa U Alice
                {
                    Tlst  {  }
                    Xlst  {  }
                    WSOC  {  }
                    RSOC  {  }
                    VSOC
                    {
                        RS1
                    }
                    VREP  {  }
                }
            }
            Xlst  {  }
            WSOC  {  }
            RSOC  {  }
            VSOC  {  }
            VREP
            {
                RS0
            }
        }
    }
}
```

B.2 Notation for versions

Resource and configuration versions can be identified by the following attributes:

- id, e.g. a filename or a name of a configuration
- Version number
- Branch name

This can be written with the following notation:

> $\mathbf{C(id)}_{\alpha}^{\beta}$ for configigurations and $\mathbf{R(id)}_{\alpha}^{\beta}$ for resources, where **id** represents the id of the resource or configuration, α is the version number and β is the branch,. If lockmode is relevant, the following notation is proposed: $\mathbf{C(id)}_{\alpha\ -\ L\ \vartheta}^{\beta}$ and $\mathbf{R(id)}_{\alpha\ -\ L\ \vartheta}^{\beta}$ where L is the lock type, e.g. W, and ϑ is the lock parameter set, e.g. { complete draft }, if present.

B.3 Notation for relations

The following notation gives neccessary information about relationships. Lockmodes and parameters of the resources and configurations are omitted.

1. **References**:
   (a) **Empty reference set**
   $\mathbf{C}(\text{bug\#23})_0^{\text{main}} \xrightarrow{r} \varnothing$
   This means that the configuration $\mathbf{C}(\text{bug\#23})_0^{\text{main}}$ holds no references.
   (b) **Configuration referencing resources**
   $\mathbf{C}(\text{bug\#23})_1^{\text{main}} \xrightarrow{r} \{\mathbf{R}(\text{a.java})_0^{\text{main}}, \mathbf{R}(\text{b.java})_1^{\text{main}}\}$
   This means that $\mathbf{C}(\text{bug\#23})_1^{\text{main}}$ references $\mathbf{R}(\text{a.java})_0^{\text{main}}$ and $\mathbf{R}(\text{b.java})_1^{\text{main}}$.
   (c) **Configuration referencing both resources and configurations**
   $\mathbf{C}(\text{nice feature})_7^{\text{main}} \xrightarrow{r} \{\mathbf{R}(\text{a.java})_{12}^{\text{main}}, \mathbf{R}(\text{b.java})_5^{\text{main}}, \mathbf{C}(\text{subfeature A})_1^{\text{test}}\ \mathbf{C}(\text{subfeature A})_2^{\text{main}}\}$
   This means that $\mathbf{C}(\text{nice feature})_7^{\text{main}}$ holds references to $\mathbf{R}(\text{a.java})_{12}^{\text{main}}$, $\mathbf{R}(\text{b.java})_5^{\text{main}}$, $\mathbf{C}(\text{subfeature A})_1^{\text{test}}$, and $\mathbf{C}(\text{subfeature A})_2^{\text{main}}$. A configuration should be able to hold references to a mix of both resources and configurations.
   (d) **Hierarchical structure**
   $\mathbf{R}(\text{c :\textbackslash rocket\textbackslash science})_0^{\text{main}} \xrightarrow{ro} \{\mathbf{R}(\text{a.java})_{37}^{\text{main}}\}$
   This means that $\mathbf{R}(\text{c : \textbackslash rocket\textbackslash science})_0^{\text{main}}$ is the hierarchical owner of $\mathbf{R}(\text{a.java})_{37}^{\text{main}}$, that is, `a.java` in revision 37 on the main branch.

(e) **Fragmentation**
$\mathbf{R}(\text{a.java})_3^{\text{main}} \xrightarrow{rf} \{\mathbf{R}(\text{a.java :: Propulsion.calculate}())_2^{\text{main}}\}$
This means that $\mathbf{R}(\text{a.java})_3^{\text{main}}$ is the owner of the fragment $(\text{a.java :: Propulsion.calculate}())_2^{\text{main}}$. A resource may own more than one fragment. This fragment notation is not a proposition, it is just to show how a fragment id may look.

2. **Version ordering**:

(a) **Revision**
$\mathbf{R}(\text{a.java})_0^{\text{main}} \xrightarrow{v} \{\mathbf{R}(\text{a.java})_1^{\text{main}}\}$

(b) **Variants**
$\mathbf{C}(\text{subfeature A})_0^{\text{main}} \xrightarrow{v} \{\mathbf{C}(\text{subfeature A})_1^{\text{main}}, \mathbf{C}(\text{subfeature A})_1^{\text{test}}\}$

(c) **Merge**
$\{\mathbf{C}(\text{subfeature A})_1^{\text{main}}, \mathbf{C}(\text{subfeature A})_1^{\text{test}}\} \xrightarrow{v} \mathbf{C}(\text{subfeature A})_2^{\text{main}}$

B.4 Chapter summary and conclusions

This chapter has presented an EBNF grammar describing allowable compositions of Xymphonic transactions as well as a notation for resources and configurations. A possible notation of their relational mappings was also described.

References

Adams, E, Gramlich, W, Muchnick, S, Tirfing, S. 1986. SunPro: Engineering a practical program development environment. In Proceedings of the International Workshop on Advanced Programming Environments (Trondheim, June), R Conradi, T M Didriksen, D H Wanvik, Eds., LNCS 244, Springer-Verlag, 86-96

Anfindsen, O J. 1997. Apotram - an application oriented transaction model. PhD thesis, University of Oslo, Norway, 1997.

Anfindsen, O J. 1994. SQL transaction management. SQL3 change proposal, ISO/IEC JTC1/SC21/WG3 DBL SOU-94.

Anfindsen, O J. 2000. Collaborative transactions. Memo on the commercial potential of Apotram, http://www.apotram.com

Anfindsen, O J, Storløpa, R. 2001. Supporting xymphonic transactions on top of Oracle. Proceedings of the International Conference on advances in infrastructure for electronic business, science and education on the Internet (SSGRR). 2001.

Anfindsen, O J. 2002. The power of Xymphonic Collaboration. Whitepaper, Xymphonic Systems AS. 2002.

Bachman, C W. 1965. Integrated Data Store. DPMA Quarterly, Jan. 1965.

Barghouti, N S, Kaiser, G E. 1991. Concurrency control in Advanced Database Applications. ACM Computing Surveys, Vol. 23, No. 3, 269-317.

Belkhatir, N, Conradi, R. 1996. SCOOP: A Unified Model for Cooperative Transactions in Software Engineering, Proc. 8th Int'l Conf. on Computing and Information (ICCI'96), 19-22 June, 1996, Waterloo, Canada.

Bernstein, P A, Hadzilacos, V, Goodman, N. 1987. Concurrency control

and recovery in database systems. Reading, Mass., Addison-Wesley.

Berzins, V. 1994. Software merge: Semantics of combining changes to programs. ACM Trans. Program. Lang. Syst. 16, 6 (Nov.), 1875-1903.

Berzins, V, Ed. 1995. Software Merging and Slicing. IEEE Computer Society Press, Los Alamitos, CA.

Binkley, D, Horwitz, S, Reps, T. 1995. Program integration for languages with procedure calls. ACM Trans. Softw. Eng. Methodol. 4, 1 (Jan.), 3-35.

Birtwistle, M G, Dahl, O J, Myhrhaug, B, Nygaard, K. 1973. Simula Begin. Petrocelli / Charter, New York, NY, 1973.

Buchmann, A, Özsu, M T, Hornick, M, Georgakopoulus, D, Manola, F A. 1992. A transaction model for active distributed object systems. In: Database transaction models for advanced applications. A Elmagarmid (ed). San Mateo, Calif., Morgan Kaufman Publishers, 123-158.

Buffenbarger, J. 1995. Syntactic software merging. In Software Configuration Management: Selected Papers SCM-4 and SCM-5 (Seattle, WA, April), J Estublier, Ed., LNCS 1005, Springer-Verlag, 153-172.

Chu-Carroll, M C. 2001. Separation of Concerns in Software Configuration Management. ICSE 2001 workshop on software configuration management (SCM 10), March 2001

Codd, E F. 1970. A relational model of data for large shared data banks. Communications of the ACM, 13, (6), 377-390.

Conradi, R, Malm, C. 1991. Cooperating transactions and workspaces in EPOS: Design and preliminary implementation. In Proceedings of the Third International Conference on Advances in Information Systems Engineering (CAISE ́91) R. Andersen, J A. Bubenko, and A. Solvberg, Eds. (Trondheim, May), LNCS 498, Springer-Verlag, 375-392.

Conradi, R, Westfechtel, B. 1998. Version Models for Software Configuration Management. ACM Computing surveys, 30, (2), June 1998.

Cronk, R D. 1992. Tributaries and deltas. BYTE 17, 1 (Jan.), 177-186.

Dahl, O J, Myhrhaug, B, Nygaard, K. 1970. SIMULA-67 Common Base Language. Technical Report N S-22, Norwegian Computing Centre, Oslo, Norway, 1970.

Dart, S. 1991. Concepts in configuration management systems. In Proceedings of the Third International Workshop on Software Configuration Management (Trondheim, Norway, June), P H Feiler, Ed., ACM Press, New York, 1-18.

Dart, S A. 1992. The past, present, and future of configuration management. Tech. Rep. CMU/SEI-92-TR-8 (july), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

Date C J. 1995. An introduction to database systems, (6th edition). Reading, Mass., Addison-Wesley.

Davies, C T. 1973. Recovery demantics of a DB/DC system. Proceedings of the ACM National Conference 28, 136-141.

Davies, C T. 1978. Data processing spheres of control. IBM systems journal, 17, (2), 179-198.

Dayal, U, Hsu, M, Ladin, R. 1991. A transactional model for long- running activities. Proceedings of the 17th International Conference on Very Large Databases - VLDB'91. Barcelona, Spain, 113-122.

Ege, A, Ellis, C A. 1987. Design and Implementation of Gordion, an Object Base Management System. Third International Conference on Data Engineering. 226-234. Los Angeles.

Eisenberg, A, Melton, J. 1999. SQL:1999, formerly known as SQL3. SIGMOD Record 28, (1), 131-138.

Elmagarmid, A K (ed). 1992. Database transaction models for advanced applications. San Mateo, Calif., Morgan Kaufmann Publishers

Elmasri, R, Navathe, S B. 1989. Fundamentals of database systems. Redwood City, Calif., Benjamin / Cummings.

Elmasri, R, Navathe, S B. 1994. Fundamentals of database systems. Redwood City, Calif., Benjamin / Cummings. Second edition.

Estublier, J, Casallas, R. 1994. The Adele configuration manager. In (Tichy 1994), chapter 4, pages 99-133.

Estublier, J, Casallas, R. 1995. Three dimensional versioning. In Software Configuration Management: Selected Papers SCM-4 and SCM-5 (Seattle,

WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 118-135.

Estublier, J. 1996. Workspace management in software engineering environments. In Software Configuration Management: ICSEÓ96 SCM-6 Workshop (Berlin, March) I. Sommerville, Ed., LNCS 1167, Springer-Verlag.

Feiler, P H. 1991. Configuration management models in commercial environments. Tech. Rep. CMU/SEI-91-TR-7 (March), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

Ford, W, Topp, W. 1996. Data Structures with C++. Prentice-Hall, Inc., 1996. ISBN 0-02-420971-6.

Fraser, C, Myers, E. 1986. An editor for revision control. ACM Trans. Program. Lang. Syst. 9, 2 (April), 277-295.

Garcia-Molina, H, Salem, K. 1987. SAGAS. Proceedings of ACM SIGMOD International Conference. 249-259.

Godart, C, Canals, G, Charoy, F, Molli, P. 1995. About some relationships between configuration management, software process, and cooperative work: The COO environment. In Software Configuration Management: Selected Papers SCM-4 and SCM-5 (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 173-178.

Goldstein, I P, Bobrow, D G. 1980. A layered approach to software desigh. Tech. Rep. CSL-80-5, XEROX PARC, Palo Alto, CA.

Gray, J N, Lorie, R A, Putzolu, G R, Traiger, L I. 1976. Granularity of locks and degrees of consistency in a shared database. In: Proceedings of IFIP Working Conference on Modelling of Data Base Management Systems. Freuenstadt, Germany. 695-723. Also in Modelling in Data Base Management Systems, G M. Nijssen, (ed), Elsevier Noth-Holland, 1976, 365-395.

Gray, J, Reuter, S. 1993. Transaction processing: concepts and techniques. San Mateo, Calif., Morgan Kauffman Publishers.

Grune, D. 1986. Distribution of the original shell script version of cvs in the comp.sources.unix volume 6 release in 1986.

Härder, T, Reuter, A. 1983. Principles of transaction-oriented database recovery. ACM Computing Surveys, 15, (4), 287-317.

Honda, M. 1988. Support for Parallel Development in the Sun Network Software Environment. Second International Workshop on Computer-Aided Software Engineering, 5-5–5-7.

Hornick, M F, Zdonik, S B. 1987. A Shared Segmented Memory System for an Object-Oriented Database, ACM Transactions on Office Automation Systems, Vol. 5, No. 1, 70-95.

Horwitz, S, Prins, J, Reps, T. 1989. Integrating non-interfering versions of programs. ACM Trans. Program. Lang. Syst. 11, 3 (July), 345-387.

Hunt, J J, Kiem-Phong, V, Tichy, W F. 1998. Delta Algorithms: An Empirical Analysis. ACM Transactions on Software Engineering and Methodology, 7, (2), April 1998, pages 192-214.

Kaiser, G E. 1995. Cooperative transactions for multiuser environments. In: Modern database systems, Kim, W (ed). Reading, Mass, Addison Wesley, 409-433.

Kaiser, G E, Feiler, P H. 1987. Intelligent Assitance Without Artificial Intelligence. Thirty-second IEEE Computer Society International Conference, 236-241. IEEE Computer Society Press, San Fransisco.

Kaiser, G E, Perry, D E, Schell, W M. 1989. Infuse: Fusing Integration Test Management with Change Management. COMSPAC 89: The Thirteenth Annual International Computer Software and Application Conference, 552-558. IEEE Computer Society Press, Orlando.

Kaiser, G E, Pu, C. 1992. Dynamic restructuring of transactions. Database transaction models for advanced applications. Elmagarmid A (ed). San Mateo, Calif., Morgan Kauffman Publishers, 265-290.

Katz, R H. 1990. Toward a unified framework for version modelling in engineering databases, ACM Comput. Surv. 22, 4 (Dec.), 375-408.

Kim, W. 1995 (ed). Modern database systems. The object model, interoperability and beyond. ACM Press / Addison Wesley Publishing Company. ISBN 0-201-59098-0.

Kjølstad, A G. 2001. Issues concerning parameter sets in Apotram. MSc thesis, University of Oslo, Norway, May 2001.

Kohler, W H. 1981. A survey of techniques for synchronization and recovery in decentralized computer systems. ACM Computer Surveys, 13,

(2), 149-183.

Korn, D, Kiem-Phong, V. 2000. Vdelta: Differencing and Compression, Practical Reusable Unix Software, B Krishnamurthy, Ed., John Wiley & Sons, Inc., 1995.

Korn, D, Kiem-Phong, V. 2000. The VCDIFF Generic Differencing and Compression Data Format. IEEE Internet-Draft, March 2000.

Kulkarni, U R, Ramirez, R G. Independently updated views. IEEE Transactions on knowledge and data engineering, 9, (5), 1997.

Leblang, D. 1994. The CM challenge: Configuration management that works. In Configuration Management, W F Tichy, Ed., Vol. 2 of Trends in Software, Wiley, New York, 1-38.

Leblang, D B, McLean, G D. 1985. Configuration management for large-scale software development efforts. In Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large (Harwichport, MA, June), 122-127.

Lie, A, Conradi, R, Didriksen, T, Karlsson, E, Hallsteinsen, S O, Holager, P. 1989. Change oriented versioning. In Proceedings of the Second European Software Engineering Conference (Coventry, UK, Sept.), C Ghezzi and J A McDermid, Eds., LNCS 387, Springer-Verlag, 191-202.

Lippe, E, van Oosterom, N. 1992. Operation-based merging. In Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments (SDE5), (Tyson's corner, VA, Dec.) ACM SIGSOFT Softw. Eng. Not. 17, 5, 78-87.

Mahler, A. Variants: Keeping things together and telling them apart. In (Tichy 1994), chapter 3, 39-69.

Merchant, A, Wu, K-L, Yu, P S, Chen, M-S. Performance analysis of dynamic finite versioning schemes: storage cost vs. obsolescence. IEEE Transactions on knowledge and data engineering, 8, (6), 1996.

Micallef, J, Clemm, G M. 1996. The Asgard system: Acitvity-based configuration management. Proc. 6th International Workshop on Software Configuration Management, volume 1167 of Lecture Notes in Computer Science, Berlin, Germany, March 1996. Springer- Verlag., 175-186.

Moss, J E B. 1981. Nested transactions - an approach to reliable dis-

tributed computing. PhD thesis, MIT Dept. of Elect. Eng. and Comp. Sci Technical Report 260.

Munch, B P. 1993. Versioning in a Software Engineering Database - the Change Oriented Way. PhD Thesis, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, Norway, Sept. 17, 1993.

Munch, B P, Larsen, J -O, Gulla, B, Conradi, R, Karlsson, E -A. 1993. Uniform versioning: The change-oriented model. In Proceedings of the Fourth International Workshop on Software Configuration Management (Baltimore, MD, May), S Feldman, Ed., (Preprint) 188-196.

Nijssen, G M. 1981. An Architecture for Knowledge base Software. Control Data / University of Brussels, July 1981.

Nodine, M, Zdonik, S. 1992. Cooperative transaction hierarchies: Transaction support for design applications. VLDB Journal, 1, (1), 41-80.

Pugh, W. 1989. Skip Lists: A Probabilistic Alternative to Balanced Trees, Proceedings of the Workshop on Algorithms and Data Structures, Ottawa Canada, August 1989.

Reed, D P. 1978. Naming and synchronization in a decentralized computer system. PhD thesis, MIT Dept. of Elect. Eng. and Comp. Sci Technical Report 205.

Rochkind, M J. 1975. The source code control system. IEEE Trans. Softw. Eng. 1, 4 (dec.), 364-370.

Silberschatz, A, Stonebraker, M, Ullman, J (eds). 1991. Database systems: achievements and opporttunities. Communications of the ACM, 34, (10), 110-120.

Silberschatz, A, Korth, H, Sudarshan, S. 1997. Database System Concepts. McGraw-Hill computer science series. McGraw-Hill, third edition. Software Maintenance And Development Systems. 1990. Aide-de-Camp Product Overwiew. Software Maintenance and Development Systems, Concord, MA.

Tichy, W F. 1982. Design, implementation, and evaluation of a revision control system. In Proceedings of the Sixth International Conference on Software Engineering (Tokyo, Sept.), IEEE Computer Society Press, Los Alamitos, CA, 58-67.

Tichy, W F. 1985. RCS-A system for version control. Softw. Pract. Exper. 15, 7 (July), 637-654.

Tichy, W F. 1988. Tools for Software Configuration Management, In Proc. of the ACM Workshop on Software Version and Configuration Control, Grassau, FRG, Berichte des German Chapter of the ACM, Band 30, 466 p., Stuttgart, January 1988, B G Teubner Verlag, pages 1-20.

Tichy, W F. 1994. Configuration Management, volume 2 of Trends in Software, Ed. John Wiley & Sons, Chichester, UK, 1994.

Walpole, J, Blair, G S, Malik, J, Nicol, J R. 1988. A Unifying Model for Consistent Distributed Software Development Environments. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 183-190. P Henderson, Ed.

Walter, B. 1984. Nested Transactions with Multiple Commit Points: An Approach to the Structuring of Advanced Database Applications. Tenth International Conference on Very Large Data Bases, 161-171. Morgan Kauffman, Singapore.

Wächter, H, Reuter, A. 1992. The ConTract model. Database transaction models for advanced applications. A Elmagarmid (ed). San Mateo, Calif., Morgan Kauffman Publishers, 219-264.

Westfechtel, B. 1991. Structure-oriented merging of revisions of software documents. In Proceedings of the Third International Workshop on Software Configuration Management, (Trondheim, Norway, June), P H Feiler, Ed., ACM Press, New York, 68-79.

Westfectel, B, Munch, B P, Conradi, R. 2001. A Layered Architecture for Software Configuration Management, IEEE Transactions of Software Engineering, vol. 27, no. 12, p. 1111-1133 (December 2001).

Whitehead, J E Jr. 1997. World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction. StandardView, Vol. 5, No. 1, March 1997.

Wieczerzycki, W. 1998. Database and Transaction Model for Dynamic and Cooperative Workflows, Journal of Computing and Information Technology, CIT 6, 1998, 1, 73-88.

Zeller, A, Snelting, G. 1995. Handling version sets through feature logic.

In Proceedings of the Fifth European Software Engineering Conference (Barcelona, Sept.), W Schäfer and P Botella, Eds., LNCS 989, Springer-Verlag, 191-204.

Zeller, A. 1996. Configuration Management with Version Sets. A Unified Software Versioning Model and its Applications. PhD thesis, Technischen Universität Braunschweig, Germany, November 1996. http://www.cs.tu-bs.de/softech/papers/zeller-phd/