Herindrasana Ramampiaro

CAGISTrans: Adaptable Transactional Support for Cooperative Work

Doctoral Thesis

Submitted in Partial Fulfilment of the
Requirements for the Degree of

Doktoringeniør

Department of Computer and Information Science
Norwegian University of Science and Technology
N-7491 Trondheim, Norway

URN:NBN:no-2103

NTNU Trondheim
Norges teknisk-naturvitenskapelig universitet
Doktoringeniør avhandling 2001:94

Institutt for datateknikk og informasjonsvitenskap (IDI)
IDI-rapport 2001:06

ISBN 82-471-5360-2
ISSN 0802-6394

URN:NBN:no-2103

Preface

This is a doctoral thesis submitted to the Department of Computer and Information Science (IDI), Norwegian University of Science and Technology in partial fulfilment of the degree of "Doktoringeniør" (Ph.D.). The work has been carried out at the Database Systems Group[1]. It was done in the context of the CAGIS – Cooperative Agents in a Global Information Space – project during the period 1997 - 2001. Part of the work was conducted during a 7 month visit to the Cooperative System Engineering Group (CSEG)[2], Computing Department, Lancaster University, UK.

Acknowledgments

This work could not be carried out without direct or indirect support and help from others. I would therefore like to express my gratitude and appreciation to all those who gave me comments, assistance and support during my work for this thesis.

My first and sincere thanks go to my adviser Professor Mads Nygård for the enormous amount of time he spent with me while doing this work. This work could not be done without his direction, guidance and encouragement. I also appreciate the opportunities he has provided me to present and discuss the ideas in this thesis through publications and conferences.

Parts of this work would not be possible without the efforts of the graduate students Tomas Holt, Lars Killingdag, Mufrid Krilic, Rune Selvåg, Pål Are Sætre, and Tor Martin Kristiansen who I advised. I would particularly like to

1. See http://www.idi.ntnu.no/grupper/db.
2. See http://www.comp.lancs.ac.uk/computing/research/cseg.

thank them for their contributions in implementing the CAGISTrans prototypes.

My former and present colleagues at the CAGIS project – in alphabetical order – Terje Brasethvik, Professor Reidar Conradi, Associate Professor Monica Divitini, Jens-Otto Larsen, Associate Professor Mihhail Matskin, Sobah A. Petersen, Professor Arne Sølvberg, and Alf Inge Wang deserve appreciation for their contributions to this work through discussion and comments. Special thanks to my neighbour and former office mate, Alf Inge, for also being a good friend.

I also would like to offer thanks to all my colleagues at the Database Systems Group – Professor Kjell Bratbergsengen, Professor Svein-Olaf Hvasshovd, Olav Sandstå, Associate Professor Ketil Nervåg, Jon Olav Hauglid, John Hegglland, Rune Humborstad, Tore Mallaug, and Maitrayi Sabaratnam. Special thanks to Associate Professor Roger Midtstraum for his valuable comments and suggestions about this work.

I further thank Professor Ian Summerville for generously hosting my visit to the Computing Department, Lancaster University, and to all members of the CSEG group for making my stay fruitful, enjoyable and interesting.

I thank all administrative and technical support staff at IDI for providing necessary infrastructure.

I also thank Joe Gorman for his contribution in proof-reading this thesis and his suggestions for improving its readability and Stewart Clark at NTNU for doing the final editing.

This work has been supported by the Research Council of Norway through grant 112567/43 via the Distributed Information Systems (DITS) programme and the CAGIS project. Their financial support is much appreciated.

I am immensely grateful for the love and support my parents and family have given me over the years. And last but not least, I would like to thank my wife Grethe, without whom nothing would be worth doing. This "ego-trip" would not be possible without her understanding and support.

Trondheim, 1 October 2001.

Herindrasana Ramampiaro.

Abstract

The theme of this thesis is on transactional support for cooperative work environments, focusing on data sharing. It is thus concerned with the provision of suitable mechanisms to manage concurrent access to shared data and resources. This subject is not new, per se. In fact, maintaining data consistency in multiuser environments is a classical problem that has been addressed thoroughly since the introduction of transaction management. However, while traditional transaction models – also called ACID (atomicity, consistency, isolation, and durability) transactions – have provided satisfactory and efficient consistency management for traditional multiuser database and business applications such as banking and flight reservation applications, they have been found to be too restrictive in the context of cooperation. Atomicity is inappropriate for cooperative environments, where activities are normally of long duration. The isolation of transactions does not allow cooperation, which is thus unsuitable for cooperative environments.

Several solutions have been proposed and developed in terms of advanced transaction models and frameworks. The goal has primarily been to overcome the limitations of traditional transactions. However, although there are many solutions, there are some problems that are not solved. Among these are the problems that result from the dynamic and heterogeneous nature of cooperative work. Finding solutions to these problems has been a subject for intensive research over the past couple of decades. However, it is widely agreed that they still deserve careful attention. The solution here is to provide transactional support that not only can be tailored to suit different situations, but can also be modified in accordance with changes in the actual environment while the work is being carried out – i.e., is adaptable. As part of this solution, we have identified and extracted the beneficial features from existing models and

attempted to extend these to form a transactional framework, called CAGISTrans. This is a framework for the specification of transaction models suiting specific applications. The main contribution in handling dynamic environments is in the way of organising the elements of a transaction model to allow runtime refinement. In addition, a transaction management system has been developed, built on the middleware principle, to allow interoperability and database independence, and support for non-database resources. This thereby addresses the problems induced by the heterogeneous nature of cooperative environments.

The solution depends on setting requirements based on the practical real-life needs for a system supporting cooperative work. This shows how this framework meets these requirements. One of the issues that is not emphasised is system throughput. From a transaction processing perspective, this issue is generally considered to be critical. However, since transactions in cooperative environments normally span long periods of time, they are mainly more sensitive to response time performance than system throughput.

A CAGISTrans system has been implemented, using several technological tools such as Java[1], XML and software agents. The thesis discusses the specific use of and general experience with these technological tools. The CAGISTrans prototypes have implemented the major parts of the framework.

The main conclusion is that the current CAGISTrans framework is able to support the basic features of dynamic and heterogeneous transaction management, allowing users to specify models and have the system execute their transactions in a flexible and controlled manner.

1. Java is a trade mark of Sun Microsystems. See http://java.sun.com.

Table of Contents

List of Figures

List of Tables

Part I

BACKGROUND AND CONTEXT

URN:NBN:no-2103

URN:NBN:no-2103

Chapter 1
Introduction

The theme of this thesis is transactional support for cooperative work environments. The focus is on developing a transactional framework called CAG-ISTrans, allowing specification and customisation of transactional models to fit specific needs. This chapter outlines the motivation for such a transactional framework, defines the problems addressed, and highlights the contributions of this work. It also describes the structure of this thesis, serving as a roadmap for the reader.

1.1. Motivation

The widespread availability and use of computers and networking – i.e., Internet and the World Wide Web (Web for short) – has undoubtedly contributed to a change in the way people carry out their work. Increasingly, work is performed in teams distributed over a network, where people get together to have the work done without strict organisational and power structures. As a result, the induced work environments are dynamic – thus the members and the structure of the teams may vary from time to time, and the imposed requirements may be in continuous change. This makes it important and challenging to provide suitable tools to facilitate cooperative activities of this type.

The field of CSCW – Computer Supported Cooperative Work – has emerged due to the need to provide appropriate computer-based tools to support group work (Batory and Kim 1985, Greif and Sarin 1987, Schmidt and Bannon 1992, Grudin 1994). The support needed ranges from informal interaction among co-workers – such as meeting and conferencing systems – to the sharing and exchange of information. Our main focus is on supporting team work within product design – Software Engineering, CAD/CAM, among others –

and product manufacturing, that are mainly based on data sharing and exchange.

Support for sharing in multi-user environments has long been a main concern of both the database and the CSCW communities (Gray et al. 1975, Gray 1981, Korth 1983, Bancilhon et al. 1985, Bernstein et al. 1987, Greif and Sarin 1987). Normally, support for sharing through the use of computers implies provisions of concurrent access to shared resources, such as databases and Web servers. However, though sharing is important, considering product design and manufacturing, the avoidance of inconsistency is crucial. This calls for the adoption of mechanisms to synchronise, coordinate and manage the aforementioned resource access. An example of a solution that has gained a considerable attention – also in the CSCW community (Ellis et al. 1991, Mariani and Rodden 1996) – originates from the database community, namely *transactions* and *transaction models*.

Note that the above-mentioned consistency may have different meanings in (traditional) database transactions and in CSCW-oriented transactions. From a traditional transaction perspective, shared data are said to be consistent if they satisfy all specified consistency constraints for the underlying database. An underlying premise is that accesses to shared data are not interfered with by any other accesses. Thus, data shared among several users can be said to be consistent only if both the final and all temporary results are considered correct (see Section 2.2 for an overview).

From the CSCW perspective, the above premise may be too strong, as simultaneous access to shared data is normal and needed. For this reason, shared data may sometimes violate some of the specified consistency constraints, but they could still be accepted as being consistent enough. The ways to reach the final results are not important as long as these results can be considered correct.

Bearing the above gap in mind, in order to successfully deal with inconsistency in cooperative work settings, it is important to find an appropriate trade-off, but this is not an easy task. In this thesis the premise is to view consistency as dependent on the requirements of the applied domain, hence stressing the provision of *adaptable* transactional support for cooperative work.

1.2. The problem with transactions

Transactions and transaction models have been widely used in managing concurrent accesses to shared data. However, the main problem with transactions – as they were traditionally intended – is their strictness. In particular, the

support for long-running activities and cooperation is missing, as a result of their atomicity and isolation requirements (Elmagarmid 1992, Conradi et al. 1997, Jajodia and Kerschberg 1997).

In order to overcome these limitations, numerous advanced transaction models have been suggested (Moss 1982, Garcia-Molina and Salem 1987, Elmagarmid 1992, Mohan 1994, Ramampiaro and Nygård 1999). In general, the main emphasis has been on the relaxation of atomicity and isolation. In terms of cooperative work, it is widely agreed that the goal has not yet been attained. For instance, many of the models were suggested with specific applications in mind, thus having fixed semantics and fixed correctness criteria. For this reason, they may fail to provide sufficient support for wide areas of applications.

One possible solution is to provide a framework for specifying and tailoring the transaction models to the application needs. This idea is not new. Most notable are ACTA (Chrysanthis and Ramamritham 1994), ASSET (Biliris et al. 1994), TSME (Georgakopoulos et al. 1996), and RTF (Barga 1999), among others. However, although these provide the ability to specify and implement extended transaction models that are suitable for specific applications, there are problems that still remain unsolved. First, support for dynamic environments – e.g., process shift – is needed but still not fully supported. One of the main reasons is that with many existing solutions, the specification of transaction models must be done before the execution of the actual transactions. Hence, they offer little or no support for adjustment during runtime. Second, cooperative work is diverse (Schmidt and Bannon 1992, Schmidt and Rodden 1996). This makes the support for heterogeneous environments highly relevant. However, the existing solutions are mainly built on DBMSs – database management systems. This means that other resource management systems other than legacy databases, such as Web servers, may not be well supported. For this reason, the heterogeneity aspect may not be adequately addressed.

1.3. Research questions

Based on the above motivation, the problems addressed in this thesis are concerned with finding appropriate ways to provide transactional support for dynamic and heterogeneous cooperative work environments.

Therefore, the main question that this thesis aims to answer is:

How can one provide transactional support that is able to deal with the dynamic and heterogeneous properties of cooperative work?

This question leads to the definition of the following subquestions, deter-

mining the development of the work:

Q1 ***Current situation:*** *are there efforts that have already answered the main question?*

Q2 ***Requirements:*** *what is the nature of cooperative work and how can it be characterised? Then, what requirements does this impose on the transactional support to be provided?*

Q3 ***Solution:*** *what foundations are necessary for the design of a system fulfilling these requirements? In other words, how can we meet these requirements for transactions?*

Q4 ***Evaluation:*** *how well do our research results solve the problems, and how does the solution compare to previous work?*

1.4. Research approach

The term research is not easily understood when looked at by itself without placing it in a context. Its meaning is nonetheless important to explore, as it is the main foundation for any doctoral work. So, what do we exactly mean by research? In general, we may see research as the process through which we reveal or discover knowledge. But, this definition is too wide, since it does not state anything about the context of the process. For instance, one may discover knowledge by measuring the air temperature, to find out what type of clothes to wear – but this process is not research per se. By contrast, if we discovered our knowledge by collecting data consisting of *multiple* air temperature measurements, resulting in statistical information – e.g., temperature variation – and its relation to current climate changes, such a process might be regarded as research. This leads us to a more generic definition – adopted from the Oxford Dictionary and Thesaurus (Tulloch 1995) – viewing research as:

> "*the systematic investigation into and study of materials, sources, etc., in order to establish facts and reach new conclusions and an endeavour to discover new or collate old facts, etc., by the scientific study of a subject or by the course of critical investigation*."

As we will see later in this section this fits well with my view of what research is.

Traditionally, research methodologies have been divided into two basic types comprising "*pure research*" and "*applied research*". *Pure research* deals with finding theories that *applied research* uses and tests in real world. However, as pointed out by Phillips and Pugh (1994), such a distinction may be too restric-

tive. For instance, applied research in many disciplines generates its *own* theories that it applies to real world experiments. This is the case with this research. Part of this work may be seen as pure research attempting to set up theories of transactions, but in addition, it is aimed at finding ways to allow use of these theories in practical "real world" cooperative applications.

Rather than the above distinctions, the nature of research towards a doctoral degree may be classified into three types (Phillips and Pugh 1994); exploratory, testing-out, and problem-solving research.

1.4.1. Exploratory research

Exploratory research is concerned with addressing new problems, issues or topics that are not well known. Thus, the idea underlying the research cannot be defined very well. But once formulated, it can be approached with theoretical investigation or empirical studies. Then, the research work will attempt to find suitable theories and concepts – developing new ones, if necessary, and deciding whether existing methodologies can be utilised or not.

1.4.2. Testing-out research

Testing-out research deals with finding the limits of previously proposed generalisations. This method thus involves a careful testing process, aimed at improving the important but incomplete generalisations developed by previous research. From this perspective, a researcher will have to designate a methodology, and apply it to a topic to which it has not been applied before, providing advantageous new knowledge and theoretical insights. Alternatively, one may apply competing theories to a new problem to determine which are best, or develop crucial experiments to make it more evident how to choose among them. A possible result of such work that can be gained is an innovative variant of an existing methodology or theory.

1.4.3. Problem-solving research

Problem-solving research uses a specific problem from "the real world" as a starting point, and finds a new methodology for its solution. In this method, the researcher will be required to discover the problem and the methodology or theory to solve that problem. In this respect, one may have to create and determine solutions addressing all aspects of a problem. Due to the complexity of real world problems, however, theories and methods may involve more than one discipline, making them more difficult to solve within the restricted bounds of a doctoral research.

1.4.4. Approach of this thesis

The research method applied in this thesis may be best classified as *testing-out research*. A problem is approached by first analysing the current situation, attempting to gather enough knowledge about existing approaches that have related solutions to the problem. This also sets up the context of the work, which is described in the first part of this thesis, including Chapters 1 (this chapter), 2, 3 and 4. As part of this, each approach has been evaluated with respect to what part of the problem it addresses, how it does this – i.e., the theory and methodology – and what problems are still necessary to solve.

The results from the first part of this thesis have revealed and extracted beneficial features of existing solutions, and led to the development of the CAGISTrans framework. This is aimed at solving the remaining problems. What these features are and how they are exploited and extended are treated in the second part of this thesis, comprising Chapters 5, 6, 7 and 8.

To evaluate this work, I have focused on the design and implementation of the framework in proof-of-concept prototypes which aim at demonstrating the capability of the framework, and test its feasibility. A comparison with related approach is also provided. This part of this work is treated in the third part of the thesis, consisting of Chapters 9, 10, 11 and 12. The results from this work were mainly measured in terms of which parts of the proposed framework could be implemented, thus providing an indication of the feasibility of this approach.

1.5. Research environment

The work presented in this thesis has primarily been conducted in the context of the CAGIS project (Conradi et al. 1996)[1]. Consequently, the work in this project is referred to as "our work" in this thesis.

CAGIS stands for Cooperative Agents in a Global Information Space. It is a basic research project in the Norwegian Research Council programme for distributed information systems (DITS). The project was devoted to provide cooperating human problem solvers – e.g., designers and engineers – with support for distributed and concurrent team work, and to develop a framework for the corresponding information technology (IT) support with distributed agents, distributed data stores, and specific formalisms and tools.

To address these objectives the project was divided into three modules concentrating on smaller more focused parts. These include handling of dis-

1. See also http://www.idi.ntnu.no/~cagis.

tributed documents and document understanding, support for cooperative processes in distributed environments, and management of sharing of distributed and heterogeneous resources. The work in this thesis covers the last module, focusing on management of sharing using transactions.

A more thorough presentation of the other modules is provided in Chapter 10, which also discusses the integration of this work into the CAGIS environment.

1.6. Publications

This thesis is partly based on papers presented at conferences published during the work that I was part of, as listed below:

- Heri Ramampiaro and Mads Nygård, "*Cooperative Database System: A Constructive Review of Cooperative Transaction Models*", In Proceedings of the **1999 International Symposium on Database Applications in Non-Traditional Environments (DANTE'99)**, pp. 315-324, IEEE Computer Society Press, Kyoto, Japan, Nov. 1999.

  This is a state-of-the-art paper, presenting a result of our initial literature studies. It reviews existing transaction models with respect to their appropriateness for cooperative work.

- Heri Ramampiaro, Monica Divitini and Sobah A. Petersen, "*Agent-based groupware: Challenges for cooperative transaction models*", In Proceedings of the **International Process Technology Workshop (IPTW 99)**, J. Estublier, G. Alonso, and H. Schlichter (ed.), Grenoble, France, Sept. 1999.

  This is a position paper following up the ideas generated from the work with the previous paper. It discusses the challenges of adopting transaction models into agent-based groupware, providing an indication of what transactional support should cover.

- Heri Ramampiaro, Alf Inge Wang and Terje Brasethvik, "*Supporting Distributed Cooperative Work in CAGIS*", In Proceedings of the **4th Annual IASTED International Conference on Software Engineering and Applications (SEA 2000)**, Las Vegas, Nevada, USA, Nov. 2000.

  This paper was written as result of a joint effort with colleagues in CAGIS. It specifically discusses the integration of the three components of the CAGIS project into a single environment supporting distributed cooperative work.

- Heri Ramampiaro and Mads Nygård, "*CAGISTrans: A Transactional Framework for Cooperative Work*", In Proceedings of the **14th International Conference on Parallel and Distributed Systems (PDCS 2001)**, pp. 43-50, Dallas, USA, Aug. 2001.

  This is an overview paper, introducing the CAGISTrans framework. It provides an initial overview and analysis of the transactional support that should be provided. It thus gives an overview of the ideas of CAGISTrans.

- Heri Ramampiaro and Mads Nygård, "*CAGISTrans: Providing Adaptable Transactional Support for Cooperative Work*", In **Proceedings of the 6th INFORMS Conference on Information Systems and Technology (CIST'01)**, pp. 3 - 29, Florida, USA, Nov. 2001.

  This is a core paper, discussing the technical foundations of our CAGISTrans transactional framework in detail. It presents its specification, design and implementation, constituting the base of this thesis.

- Heri Ramampiaro and Mads Nygård, "*Supporting Customisable Transactions for Cooperative Work: An Experience Paper*", In Proceedings of the **2002 Western Multi conference (WMC'02) – Collaborative Technologies Symposium 2002 (CTS'02)**, San Antonio, USA, Jan. 2002. (To appear).

  This is an experience paper, discussing the implementation of a CAGISTrans prototype. It discusses and evaluates our framework with respect to requirements for cooperative work. It also outlines our experience from the implementation of the prototype and discusses further issues that must be taken into account.

1.7. Contributions

The main contributions of the thesis are as follows:

- *Provision of customisable transactional models with integrated workspace support.* The work develops a transactional framework with both a possibility to customise transaction models to specific applications and integrated workspace support to increase the support for controlled sharing of resources.

- *A useful and effective way to organise vital elements of transaction models.* A transactional framework identifying elements that transaction models should provide is proposed. This includes how these are organised to

improve the transactional support for cooperative work. Fundamental to this is distinguishing between characteristics and execution specifications of transaction models. In this way, we gain an increased modularity by allowing component-based specification. This distinction also allows a user[1] to distinguish between *design time* and *runtime* specifications of transaction models, thus achieving an increased flexibility and increased support for dynamic environments.

- *A new way to support runtime management of transactional behaviour*. This work is aimed at developing an efficient way to manage runtime specification of transactional behaviour. This further improves the support for evolution in cooperative work processes by enabling users to fit the behaviour of their transactions to their current needs. Because of this, the need for a complete a priori knowledge of actions to be carried out is strongly reduced, thus improving the transactional support for unpredictably. Further, users have better control over the execution of tasks, making them able to choose appropriate actions based on current needs.

- *Analysis and development of dynamic user re-definable correctness constraints.* A transactional framework is proposed that not only allows users to specify suitable correctness constraints but also refine these at runtime. Hence, the proposed framework allows users to define correctness constraints in accordance with the needs of their applications. As a result, we can gain an increased flexibility and increased support for cooperation. Further, the ability to refine these constraints during runtime makes it possible to meet new requirements while the work is in progress.

- *Development of a transactional framework architecture supporting heterogeneous systems*. The CAGISTrans framework is designed and implemented in a system built on the middleware principle, providing not only advanced transactional support to various types of applications, but also running on a variety of resource management systems, including Web servers and legacy databases.

- *Application of a specification language making use of XML.* The development of our CAGISTrans framework exploits the extensibility of XML in a new and useful way to enable the specification of transaction models and adjust these at runtime.

1. A user is normally a transaction model designer.

1.8. Organisation of the thesis

This thesis consists of twelve chapters divided into three parts, organised as follows:

- *Part I* provides the setting of the thesis, outlines the background and context and reviews previous relevant work.
  - *Chapter 1* (this chapter) describes the background and context for the work. It also outlines the problem that the work is addressing.
  - *Chapter 2* provides an overview of basic concepts underlying this work. It focuses on the background theory of transaction processing.
  - *Chapter 3* outlines the technology relevant to this work and discusses how these relate to the work.
  - *Chapter 4* is a state-of-the-art chapter and thus provides a survey of existing solutions, focusing on transactions. Together with Chapter 2, it attempts to give an initial answer to research question Q1 (see Section 1.3).
- *Part II* discusses the requirements for cooperative work, introduces the CAGISTrans framework, and discusses the design and architecture of the framework.
  - *Chapter 5* analyses the characteristics of cooperative work and the imposed requirements. It thus attempts to answer research question Q2.
  - *Chapter 6* presents the CAGISTrans framework, discussing and analysing the contributions of this work.
  - *Chapter 7* outlines the theory, the specification and the design of the CAGISTrans framework to solve the problems arising from the dynamic nature of cooperation environments. This chapter thus attempts to answer research question Q3 (see Section 1.3).
  - *Chapter 8* provides a presentation of the design of a CAGISTrans system architecture. Thus, it particularly shows how the CAGISTrans framework deals with problems incurred by the heterogeneous nature of cooperative environments, in addition to those incurred by their dynamic nature. Therefore, this chapter further answers research question Q3.

- Part III discusses the realisation of the CAGISTrans system, evaluates the thesis and concludes this work.
  - *Chapter 9* discusses the realisation of a CAGISTrans system in a proof of concept prototype. It uses the results from Chapter 7 and Chapter 8, serving as a test-bed for the transactional framework.
  - *Chapter 10* evaluates the CAGISTrans framework. It discusses several issues that have been taken into account. It also provides a critical assessment of our solution and compares this with previous work. It is thus aimed at answering research question Q4 (see Section 1.3).
  - *Chapter 11* presents how our framework is integrated in the CAGIS environment. It also describes in a practical scenario how the CAGIS modules interact.
  - *Chapter 12* concludes the work, summarising its contributions and suggesting directions for future research.

Work in this thesis is developed based on a set of research questions and two sets of requirements. They are treated in this thesis as follows:

- *Research questions* [*Q1 – Q4*]. These research questions are introduced in Section 1.3. They are implicitly addressed throughout the thesis, but explicit, consolidated answers are provided in Section 11.2.2.
- *Transaction requirements* [*TR1 – TR7*]. These consist of a set of requirements that cooperative work imposes on transactional support. They are introduced in Section 5.3 and are implicitly addressed through the development of the CAGISTrans transactional framework. An explicit, consolidated treatment is given in Section 11.2.1.
- *System requirements* [*SR1 – SR5*]. These include a set of requirements used as bases for the development of a CAGISTrans system architecture. They are presented in Section 8.2 and explicitly addressed in Section 8.6.

To illustrate the ideas in this work, two sets of scenarios are included:

- The first scenario is a *software development* scenario, used as basis for explaining the ideas of our CAGISTrans framework. This scenario is presented in Section 5.1 and applied in Chapters 5 and 7.
- The second scenario is a *conference management* scenario, used to illustrate the integration of the three CAGIS components and their

functionality. This scenario is presented in Section 10.3 and applied in Section 10.4.

A list of main contributions is presented in Section 1.7, discussed throughout the thesis and then again summarised in Section 12.2.

Chapter 2
Basic Transaction Concepts

This chapter introduces the basic concepts of database transactions. Its main goal is to provide basic understanding of transactions, which are the basis for the work in this thesis.

2.1. Introduction

A *database* is a collection of persistent data objects satisfying a set of integrity constraints. For example, in a flight reservation database, an integrity constraint may be that no seat can be reserved for more than one passenger and that a data item containing the number of passengers for each flight is equal to the sum of the number of seats reserved. A *database management system (DBMS)* is responsible for coordinating all access to the database. This implies that all database manipulation must be performed by means of its DBMS. Users interact with the DBMS by executing special application programs. Such an execution results in a partial ordered set of read and write operations, called *transactions* (Bernstein et al. 1987). Broadly speaking, a transaction is thus the set of operations that access and change the content of a database, together with transactional commands – i.e., *begin, commit* and *abort* – marking its boundaries. In other words, a transaction may be regarded as a program segment starting with a *begin* command and ending with a *commit* (or *abort)* command. A commit command indicates that the execution of the transaction was successful, and thus all its updates should be incorporated into the database. An abort commands means that the transaction has failed, and hence all its effect should be cancelled or abolished by the DBMS.

2.2. Transaction model

2.2.1. Properties of a transaction

A primary goal of a DBMS is to allow several user transactions to concurrently access an underlying database. It thus ensures that each transaction that updates the objects of the database always preserves the integrity constraints of the database. In addition, a DBMS must protect user programs from hardware and software failures. To achieve this, transactions are constrained by the following four fundamental properties, also known as the ACID properties (Gray 1981, Härder and Reuter 1983, Bernstein et al. 1987).

- *Atomicity* referring to the fact that a transaction must be executed successfully or it appears as if it had not been executed at all – i.e., *all or nothing*. This means that all operations of a transaction must be executed, or none of them. Thus a transaction must be treated as an indivisible unit of work.

- *Consistency* requiring that each executed transaction always preserves the consistency of the database. A transaction must transform a database from one consistent state to another consistent state. This means that each successful transaction by definition only commits legal results.

- *Isolation* referring to the fact that a transaction must not observe the intermediate results of other transactions. This means that a transaction must not make its modification visible to other transactions until it has successfully completed.

- *Durability* requiring that once a transaction has committed, all its updates become permanent in the database. This means that once a transaction has successfully reached its end, none of its results are forgotten by the DBMS. All committed results must be able to survive any type of subsequent failure.

The above requirements imply that individual transactions must be programmed in such a way that they always preserve the consistency of the database. The DBMS must, in addition, ensure that each of the properties is guaranteed for *interleaved* executions of transactions. This is achieved by a combination of two different sets of protocols (Bernstein et al. 1987, Gray and Reuter 1993): (1) *concurrency control protocols* and (2) *recovery protocols*, outlined below.

2.2.2. Concurrency control

The main purpose of concurrency control is to deal with anomalies that arise when executing two or more transactions in parallel. It is reasonable to assume that in a multiuser database system several transactions will be active simultaneously. Uncontrolled execution of these transactions is likely to cause problems. Typical examples are dirty reads and lost updates (Gray and Reuter 1993), illustrated in Figure 2.1. Dirty reads occur when a transaction T_1 reads an object x previously written by another transaction T_2 which later aborts. With lost updates, a transaction T_2 writes to an object x that is ignored by another transaction T_1, which writes to this object based on its original value.

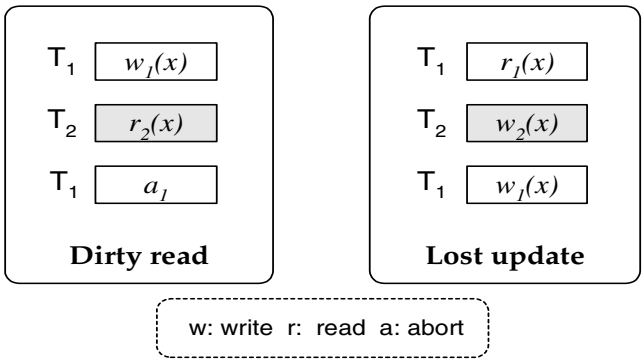


Figure 2.1 Examples of concurrency anomalies.

To deal with these problems, traditional concurrency control protocols are applied to ensure that concurrent executions of transactions have the same effect on the database as some *serial* execution of the transactions. This is known as *serialisability* (Bernstein et al. 1987). An execution is serial if and only if for every pair of transactions, all the operations of one transaction are executed in sequence before all the operations of the other.

2.2.2.1. Correctness criterion: the serialisability theory

Serialisable execution is necessary to satisfy the isolation property of transactions. The idea behind serialisability is that if individual transactions can be assumed consistent, serial execution of a set of transactions must be correct. Therefore, since each serialisable execution of concurrent transactions has the same effect as one or another serial execution, such executions are correct.

There are two typical examples of serialisability; view serialisability and conflict serialisability. An execution of a set of concurrent transactions is said to be *view serialisable* if there exists a possible serial execution of the same set such

that each transaction reads the same values and the final values of all data items in the database are the same.

Unfortunately, ensuring view serialisability has been proven to be an NP-complete problem – i.e., a computationally infeasible problem (Papadimitriou 1979). This means that finding an efficient algorithm for view serialisability is hard. For this reason, *conflict serialisability*, defined by means of conflicting operations – i.e., *read* and *write* operations, has been developed. In general, two operations are in *conflict* if they belong to two different transactions and the order in which they are executed matters. This means that with read and write operations of different transactions, two operations conflict if both of them are on the same object and at least one is a write operation (Bernstein et al. 1987). For example, for two transactions T_1 and T_2, $write_1(x)$ and $read_2(x)$, $read_1(x)$ and $write_2(x)$, and $write_1(x)$ and $write_2(x)$ all conflict. Now, an execution is said to be conflict serialisable if the order of each pair of conflicting operations is the same as that in one or another serial execution. See Figure 2.2 for an illustration of this. Note that any conflict serialisable execution is also view serialisable, but not vice versa.

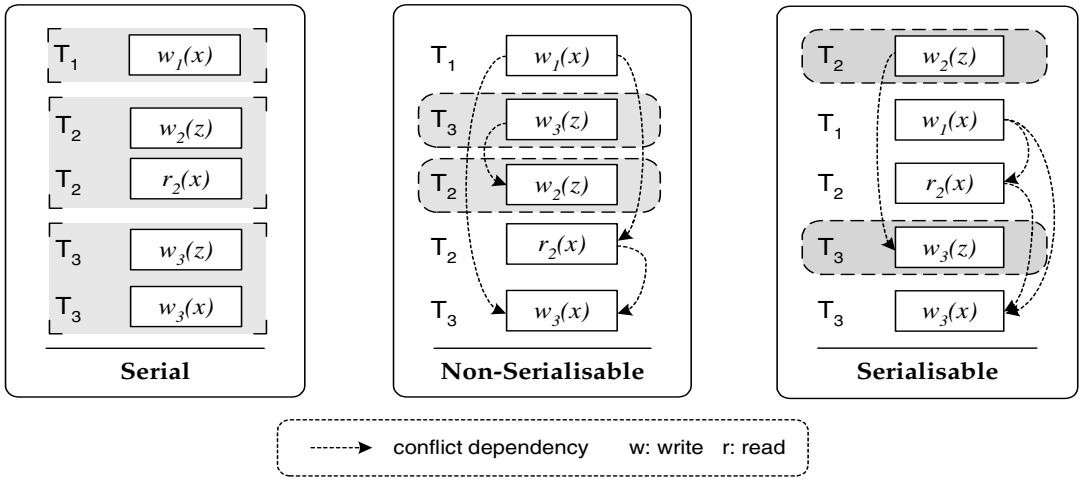


Figure 2.2 Illustration of a serialisable and a non-serialisable executions.

A simple way to verify that an execution is conflict serialisable is to use a *serialisation graph* (Bernstein et al. 1987). This is a directed graph that is constructed based on the prevailing conflict dependencies among transactions. A transaction T_i is said to depend on another transaction T_j if there is an operation of T_i that precedes and conflicts with another operation of T_j. Based on this, the nodes of the serialisation graph are transactions, while the edges are the dependencies. It has been shown that an execution is conflict serialisable iff its serialisation graph does not contain any cycle (Bernstein et al. 1987). To il-

lustrate this, the serialisation graphs for our examples in Figure 2.2 are depicted in Figure 2.3. The first execution is not serialisable as its serialisation graph contains a cycle. Conversely, since the graph for the second execution does not contain any cycles, it is serialisable.

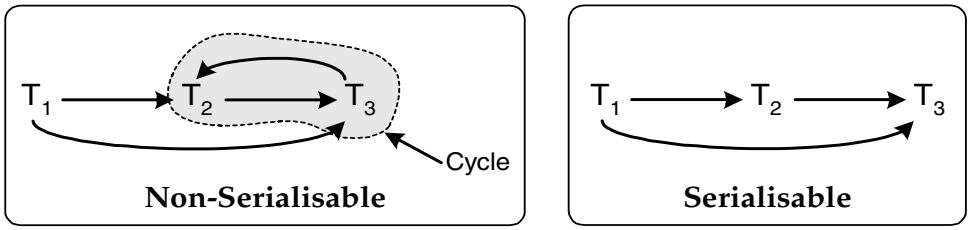


Figure 2.3 The serialisation graphs for the executions in Figure 2.2.

2.2.2.2. Concurrency control protocols

There are several efficient concurrency control protocols that can ensure serialisable executions. These can be classified as *pessimistic* and *optimistic* protocols. *Pessimistic* protocols require a transaction to ask for permission before it can perform an operation. Locking protocols, such as *2-phase locking (2PL),* are in this category. Here, locks represent the permission to perform operations. In the case of potential conflicts, a transaction requesting a lock will have to wait until the conflicting lock is released. Otherwise, the lock request is granted. The 2PL requires a transaction to acquire all its locks before it releases any lock. It may thus suffer from deadlocks since cyclic waits may occur – i.e., two or more transactions wait for each other forever. Another drawback is long duration blocking, that is likely to occur when transactions are long-running. A non-locking pessimistic protocol that avoids deadlocks is *timestamp ordering* (Bernstein et al. 1987).

Another possible way to deal with both blocking and deadlocks is to allow transactions to perform their operations without requiring them to ask for permission first. Instead, when they attempt to commit, the transactions are validated to make sure that their executions are serialisable. If this fails, one or more transactions will be forced to abort. Such protocols are called *optimistic* protocols. The main drawback with this is that a lot of work could be thrown away if the execution was not serialisable. For this reason, pessimistic protocols are often preferred.

2.2.3. Recovery concepts

Recovery protocols are aimed at dealing with failure anomalies (Bernstein et al. 1987, Gray and Reuter 1993). Thus, their primary purpose is to enforce the

atomicity and durability properties of transactions. Along this line, the underlying basic idea of the recovery protocols is that the DBMS has to make sure that the database contains (1) all of the effects of committed transactions and (2) none of the effects of aborted ones.

There are three typical categories of failures that recovery protocols must handle (Bernstein et al. 1987, Gray and Reuter 1993):

1. *Transaction failure*. This type of failure occurs as a result of faults relating to a specific transaction, and does not involve any loss of storage. Such faults imply that a specific transaction aborts. They are detected and enforced by the transaction – such as finding wrong values when accessing data items – or by the system – such as selecting the transaction as a victim to resolve a deadlock. To handle transaction failures, the DBMS executes *undo actions* (see Section 2.2.3.2).

2. *System failure*. This type of failure occurs due to loss of volatile storage such as primary memory. It could be caused by power loss or the like. System failures only affect the volatile storage, while data – e.g., the system state – in non-volatile storage, such as disks, survives system failures. To handle system failures, the DBMS must make sure (1) that the effects of transactions that were in a committed state are incorporated into the database, and (2) that the effects of all active or aborted transactions at the time of the system failure are wiped out from the database. For this reason, the DBMS executes *undo* and *redo actions (*again see Section 2.2.3.2).

3. *Media failure*. This type of failure occurs because of a loss of non-volatile storage such as disk memory. It could be caused by disk head crashes. It may result in a loss of the current database only or even the whole database, including the current database and diverse logs (see Section 2.2.3.2). Because of this, all transactions will be affected, and the damage may in some cases be hard to repair. In this type of failure, the DBMS executes *redo actions*.

2.2.3.1. Recoverability, cascading aborts and strictness

A primary requirement of recovery protocols is that concurrent executions of transactions must be *recoverable* in addition to being serialisable (Bernstein et al. 1987). An execution is recoverable if any transaction that reads values written by a transaction T only commits after T has committed. In other words, if T aborts, all transactions that have read values written/modified by T must abort too. These aborts may in turn lead to abortions of other transactions

and so on. This phenomenon known as *cascading aborts* – i.e., abortion of T_1 implies the abortion of T_2, then T_3, etc.

The simplest way to avoid such a situation is to require transactions to only read committed data. Thus, T_j may read values written by another transaction T_i only if T_i has committed. Otherwise, T_j must wait. Executions following this rule are referred to as executions that *avoid cascading aborts* (Bernstein et al. 1987).

This far, it suffices to restore the original value of an object, when the transaction that has changed the object value aborts – i.e., restoring the *before image* value. However, this could still cause problems if abortions appear after two or more consecutive writes. This means that restoring the before image may result in incorrect final values. For this reason, an additional restriction is necessary. This requires transactions to only read or modify data written by other transactions that have either committed or aborted. Executions satisfying such a restriction are known as *strict* executions (Bernstein et al. 1987).

2.2.3.2. Undo and redo actions: logging

Recoverability, avoid cascading aborts and strictness only concern transaction failures. To handle system and media failures too, recovery protocols try to make sure that the database system can be brought back to the most recent state – i.e., the latest consistent state – it had before the failure occurred. For this to be possible the protocols must address the usage of volatile buffers and different strategies to propagate updates to the permanent database.

A failure often causes the state of the system in volatile buffers to be lost, while the state in the permanent database may survive. Recovery from such failures can be accomplished by two basic actions (Bernstein et al. 1987):

(1) *Undo action*, eliminating the effects of updates on the database done by a "failing" transaction. This action is needed for *transaction failures* and *system failures* – i.e., due to the presence of uncommitted data in the database.

(2) *Redo action*, reprocessing updates on the permanent database done by a "successful" transaction. This action is needed for *system failures* and *media failures* – i.e., due to the absence of committed data in the database.

Bookkeeping must be performed to successfully use these actions. This is kept in the *database log* in stable storage. The log keeps track of all transactions that have committed and aborted, and all operations that have read and modi-

fied data objects in the database. Thus, the above actions, together with the log are often used for a system to recover from system and media failures.

2.3. Transactions and advanced applications

The usefulness of transactions in multiuser environments is evident. Transactions allow well-defined correctness criteria and efficient failure handling. Thus, they achieve consistent and reliable data management.

However, the increased use of transactions in non-traditional database environments challenges the traditional transaction model. In particular, the atomicity and isolation properties have two implications that are rather undesirable in advanced applications. First, they limit cooperation. We recall that data must be kept private or locked by a transaction until it is terminated. Hence, other transactions will be forced to wait for this termination. This prevents data from being freely exchanged among cooperating parties and made accessible as soon as needed. A possible solution is to relax the isolation requirement, thus allowing parties to share their intermediate results. Second, if a transaction fails – due to a transaction failure, a system failure or a media failure – all work performed inside its context must be abolished. As a result, a lot of invested effort could be wasted, although not directly influenced by the causes of failure. Moreover, cascading aborts may occur if cooperation took place with the transaction that has failed, and this is unacceptable. A possible way around is to allow a transaction to abort its private work at a fine-grained level, and to restart it without causing *related* but unaffected cooperating transactions to suffer.

The main conclusion is that traditional ACID transactions are inappropriate for advanced (cooperative) applications. For this reason, efforts in developing solutions to make use of the transaction concept in these applications have gained an increasing relevance. Early work in extending the traditional transaction model kept all the ACID properties, but added a nesting structure to allow fine-grained recovery management and increased intra-transaction concurrency (Moss 1982). Other approaches took advantage of semantic knowledge of database operations (Korth 1983, Weihl 1988, Badrinath and Ramamritham 1992) to reduce the probability of conflicts and hence to increase the level of concurrency. However, they still do not provide adequate cooperation support.

Newer efforts have attempted to find efficient ways to compromise on the atomicity and isolation properties but at the same time preserve consistency and durability properties (Elmagarmid 1992, Mohan 1994, Kaiser 1994). The

work presented in this thesis attempts to follow up these efforts, addressing problems in modern environments, characterised by their dynamic and heterogeneous natures. Satisfactory solutions to these problems are still absent in existing transaction models. Chapter 4 further reviews existing transaction models developed for advanced applications.

Chapter 3
Technological Overview

The goal of this chapter is to provide a background survey of the technological platforms underlying the realisation of the ideas of this thesis. First, Section 3.1 gives an overview of computer supported cooperative work, mainly focusing on groupware. It outlines the categories of existing groupware applications through a set of existing classifications. Section 3.2 outlines the notion of middleware, providing an overview of different middleware classes and stating the relation of middleware to this work. Finally, Section 3.3 gives an overview of agent technology, stating its role in the work presented in this thesis.

3.1. Computer Supported Cooperative Work

Computer supported cooperative work – CSCW for short – has existed as a research field since the early eighties. The CSCW acronym was coined by Chasman and Greif as a theme of a workshop that they organised in 1984. Their goal was to invite people from different fields, sharing interests in understanding how people work and the role of technology in their work environments. One of the main motivations for this initiative was the fact that the technological aspect of approaches to group support, such as office automation, were no longer the only challenge. To provide satisfactory support, technologists needed to learn how people work in groups and organisations, and how technology affects that (Grudin 1994).

Soon after its introduction, CSCW became a multi-disciplinary field, aiming at examining how groups of people work and attempting to find a way to provide appropriate technological (computer) support for the work process. Although since its origin CSCW has been criticised for being too vague due to

its lack of clear focus (Schmidt and Bannon 1992), the field is now well established, with annual international and European conferences – e.g., the ACM CSCW and ECSCW conferences – and a variety of journals.

3.1.1. Groupware

Given the rapid evolution of computer technology, it is still necessary to focus on the technical aspect of CSCW. The part that deals with CSCW applications is referred to as groupware (Grudin 1994). Thus, while CSCW is concerned with the theoretical foundations and methodologies for teamwork and its computer support, groupware is the software system supporting teamwork and integrating theoretical foundations achieved by CSCW research (Borghoff and Schlichter 2000). Early experiments with groupware applications have resulted in various classes of tools. Examples of these typically include desktop and video conferencing systems, collaborative authoring applications, electronic classrooms, and group support systems, among others.

3.1.1.1. Advantages of groupware systems

From a cooperative work perspective, the usefulness of groupware is evident. First, they facilitate communication by making it faster, clearer and more persuasive. Second, they support communications where they would not otherwise be possible. Third, groupware systems enable telecommuting and thus cut down on travel costs. Fourth, since sharing is possible, they bring together multiple perspectives and expertise, and form groups with common interests where it otherwise would be impossible to gather a sufficient number of people face-to-face. Finally, groupware systems make it possible to save time and costs in coordinating group work and facilitating problem solving.

There are several groupware products that have attempted to exploit these advantages. As yet, only a few have enjoyed the commercial success of Lotus Notes[1] (Notes for short), which now has more than 55 million users (Mohan et al. 2000). Notes is a groupware system consisting of several modules, including a client-sever messaging system, calendaring, scheduling, high-level workflow process definition, a ODBC driver for accessing external databases, and conferencing facilities (Papows 1995). From the database perspective, Notes provides an extensive support for semi-structured data, which also allows the exchange and manipulation of unstructured information. It is also one of the earliest groupware systems that have provided support for data replication. Further, Notes provides traditional transactional support with sophis-

1. Lotus Notes is a trade mark of the IBM Lotus Corporation.
See http://www.notes.net.

ticated recovery management, using the ARIES log-based recovery methods (Mohan and Narang 1994, Mohan et al. 2000). Notes provides database access that can be characterised by high availability and a certain degree of resource heterogeneity. Since cooperative transactional support is not emphasised in Notes, it is in this sense different from the approach of CAGISTrans. Also the scope of the present work is not developing a groupware, per se. Rather, the focus here is on the provision of the transactional support for the cooperative activities. In a sense, such a support could be integrated in a groupware system.

3.1.1.2. Classification of groupware systems

The classification and characterisation of groupware systems or applications has been argued to be useful for the development of such groupware applications (Ellis et al. 1991, Grudin 1994). In particular, it can be used to identify applications that pose common technical challenges – e.g., those dealing with concurrent activities. Since the birth of groupware, there have been a variety of classifications of groupware applications. One of the earliest, which is perhaps the most familiar, is the time/space taxonomy (DeSanctis and Gallupe 1987). This taxonomy classifies groupware applications into four classes according to *time* – i.e., synchronous and asynchronous collaboration modes, and *space* – i.e., same and different geographic locations. This has since been extended by other researchers to cover other dimensions of collaboration types. An example is that proposed by Grudin (1994) depicted in Figure 3.1. He extended the original taxonomy with an extra measure along each dimension indicating whether

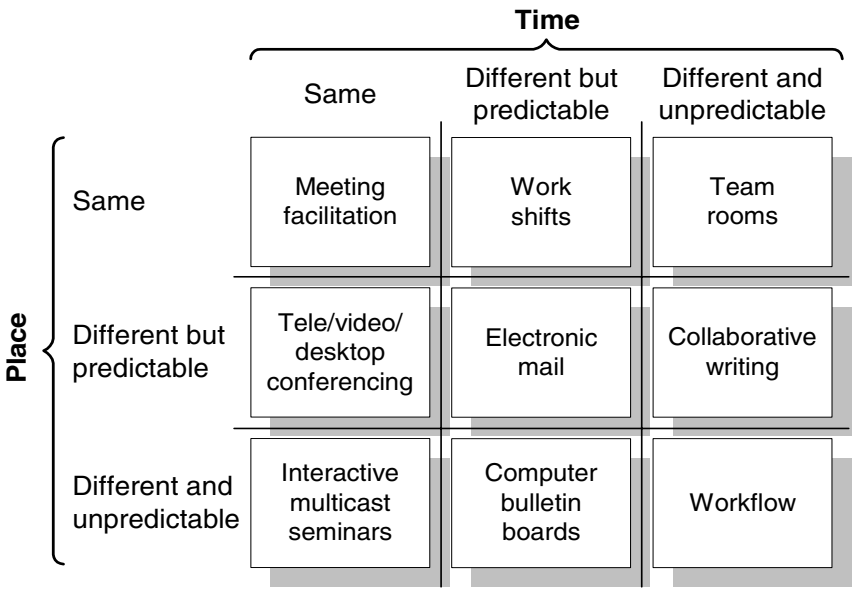


Figure 3.1 Grudin's (1994) extended time/space groupware typology.

the modes and locations of the collaboration are predictable or unpredictable. Activities can be carried out at different times that are highly predictable or constrained. An example of this is sending e-mail to a colleague, where one may expect him/her to read the message within a day or so (see middle column in Figure 3.1). Alternatively, activities can be performed at different times that are unpredictable, as in open-ended collaborative writing projects (see right-hand column in Figure 3.1).

Another extension based on the time/space matrix was proposed by Rodden (1991). In contrast to Grudin's (1994) – i.e., instead of adding extra dimensions, he presented an application level classification, identifying applications having more than a fixed time or space characteristics. See Figure 3.2, a reproduction of (Rodden 1991), for an illustration. For example, people engaged in activities involving the use of a multimedia conferencing systems may be co-located or distributed – i.e., some people may be co-located while others are distributed, and so on.

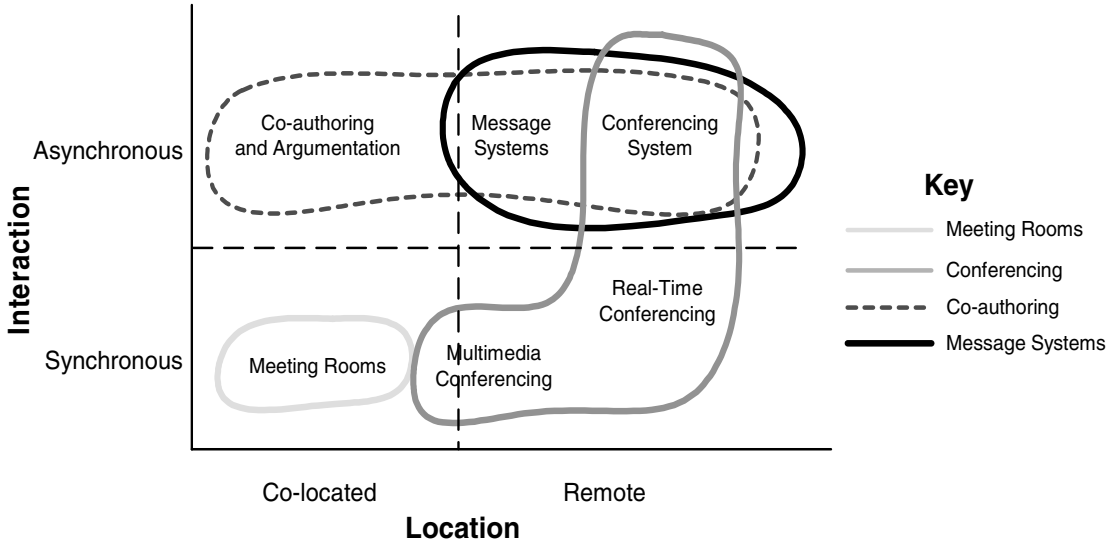


Figure 3.2 Rodden's (1991) groupware classification.

The time/space classification is incomplete by far. In fact, it has been criticised by some researchers due to this incompleteness (Schmidt and Rodden 1996, Greenberg and Roseman 1998). In particular, the necessity of seamless transition among the different dimensions is stressed. Teamwork often shifts easily from being asynchronous to being synchronous, or from co-located to distributed. Therefore, to successfully capture these changes, classifications have to take into account the fact that groupware must deal with many types of cooperative processes and modes, and facilitate seamless transition from one

type of cooperation to another. To further emphasise this, Greenberg and Roseman (1998) claimed that many systems developed based on the time/space taxonomy may fail to cover all aspects of real-world cooperative work processes.

A classification by Ellis et al. (1991) seems interesting from this perspective. They proposed two classification dimensions that are continuous, and do not consider an explicit distinction between synchronous and asynchronous cooperation. Also they do not explicitly consider the locations of cooperating individuals. As depicted in Figure 3.3, the two dimensions considered are *common task*, identifying the supported degree of task sharing, and *shared environment*, identifying provided information about the participants, the current state of projects and the social atmosphere etc. (Ellis et al. 1991). As an example, a timesharing system such as a traditional database lies at the low end of the common task dimension. Although such a system supports several users performing concurrent tasks, these tasks are mainly separate and independent. On the other hand, the CSRS – collaborative software review system (Tjahjono 1996) allows a group of designers to evaluate a software module during a runtime interaction and thus lying at the high end of the common task dimension. Further, most electronic mail systems support the notion of common task, but they do not support a shared environment, in the way that, for instance, electronic classroom systems (Shaw 1995) do.

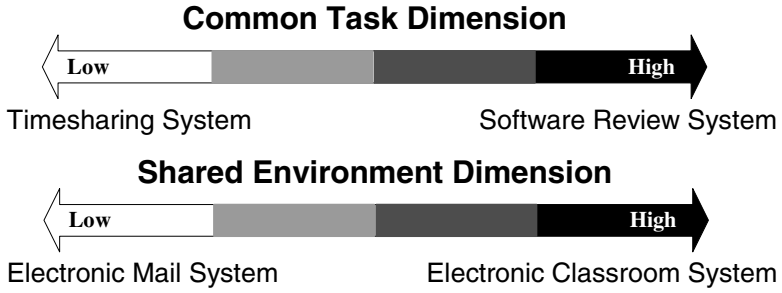


Figure 3.3 Groupware classification dimensions according to Ellis et al. (1991).

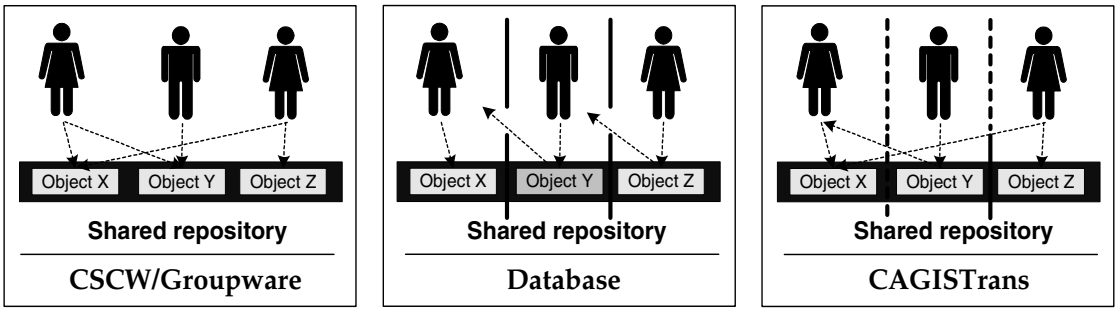
Other classifications also exist. Examples are (Greenberg and Roseman 1998) and (Farshchian 2001). In Chapter 5, we will discuss cooperative work characteristics that are relevant for the development of transaction models supporting cooperative work.

3.1.2. CSCW related to this work

This work can be regarded as an intersection between database and groupware. This means that the focus is on the technological aspect of cooperative work. It also recognises the importance of understanding the way people

work in their environment. But, it is beyond the scope of this work to do a study of this. Instead, the focus is on the way to make the developed system flexible and adaptable enough to cover the widest possible situations and application areas.

In general, the main emphasis of groupware systems is the notion of "what you see is what I see" – WYSIWIS for short. This means that in addition to sharing, an important issue that groupware systems have focused on is the way to provide efficient infrastructures and user interfaces, making participants aware of other people and events that are relevant to their tasks (see Figure 3.4a). The literature calls this awareness (Ellis et al. 1991, Gutwin et al. 1996, Greenberg and Roseman 1998, Farshchian 2001). From this perspective, the underlying philosophy of groupware contradicts with that of traditional database systems. Database systems strive to enforce isolation – cf., Chapter 2, providing their users with the illusion of operating alone (see Figure 3.4b). Thus, awareness is de-emphasised. Further, while most cooperative applications leave the control of concurrency to the collaborating individuals to find out – though its importance has been advocated by some groupware researchers (Ellis et al. 1991, Greenberg and Marwood 1994) – database systems stress the necessity of providing transparent system-based concurrency control (Bernstein et al. 1987). The former may introduce anarchy. Thus, consistency of resources cannot be guaranteed. By contrast, the latter – isolation – could be just a burden since sharing is not possible. Bearing this in mind, the work in this thesis is aimed at finding a way to bridge these gaps. This means finding ways to provide "transparent walls" among users, allowing them to be aware of others, and enabling the controlled sharing of resources among them (see Figure 3.4c).



(a) (b) (c)

Figure 3.4 Illustration of the multiuser support alternatives.

3.2. Middleware

Middleware is a notion that covers a very broad area of distributed com-

puting. Because of this, an exact formal definition is hard to find. Some define middleware as a software component that connects two otherwise different applications[1]. Others view middleware as the component of distributed systems that addresses the scalability issue by providing a distributed infrastructure layer supporting applications.

3.2.1. Middleware classes

Despite the lack of exact definition, many agree that middleware is capable of dealing with the impact of problems related to the development of applications in heterogeneous environments. This is achieved by providing an isolation layer of software that relieves application developers of handling such heterogeneity themselves, by presenting their own enabling layer of APIs – application programming interfaces. This layer decouples applications from any dependencies on platform specific APIs. Hence, heterogeneity becomes transparent to developers (Bernstein 1996, Spahni et al. 1998).

Five commonly known middleware categories are:

1. *Remote Procedure Call (RPC)* based middleware, providing interfaces allowing applications to execute functions on a remote host in a transparent way – i.e., as if these functions were local. RPC provides developers with the possibility to define their functions using an interface definition language (IDL). Then these functions are compiled into client and server stub code that does the networking. Examples of existing RPC products include the Sun Microsystems' RPC[2] and NobleNet's RPC[3].

2. *Message Oriented Middleware (MOM),* offering a basic set of network commands such as SEND and RECEIVE, to allow programs to send and receive data to and from other programs in real time. MOM is analogous to e-mail in the sense that it is asynchronous. It requires the recipients of messages to interpret their contents and meaning, and take suitable actions based on this interpretation. Examples of products within this category include BEA's MessageQ system[4] and IBM's MQSeries[5].

3. *Distributed Transaction Processing (DTP) monitors*, providing an environment for handling transactions over a network. DTP monitor systems

1. See http://webopedia.internet.com/TERM/m/middleware.html.
2. See http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1057.html.
3. See http://www.geibelpr.com/noblenet.htm.
4. See http://www.bea.com/products/messageq.
5. See http://www-4.ibm.com/software/ts/mqseries.

are often built on top of MOM and RPC technology, but also offer control and management functionality. Examples of existing DTP monitors include Transarc's Encina[1] and BEA's Tuxedo[2].

4. *Database Oriented Middleware (DOM)*, offering APIs to facilitate transparent accesses to database systems. DOM systems often use widely accepted APIs such as ODBC – open database connectivity – and JDBC – Java database connectivity. Using ODBC or JDBC, applications access remote data using structured query language (SQL). Several commercial products are available supporting this API, including Oracle's SQL*Net[3], Sybase's Open Client[4]/Open Server[5], among others.

5. *Object Request Brokers (ORBs*), enabling objects that comprise an application to be distributed and shared across heterogeneous networks. ORB systems provide interfaces that relieve application programmers from having to create the links among objects and functions or other APIs offered by other middleware solutions. Today, there are several ORB specifications. Examples of these are OMG's common object request broker architecture – i.e., CORBA (Object Management Group 1998) and Microsoft's object linking and embedding – i.e., OLE (Brockschmidt 1995). Examples of products supporting the ORB middleware include IONA's Orbix[6] and Microsoft's distributed component object model DCOM[7].

Refer to (Bernstein 1996) and (Spahni et al. 1998) for detailed treatment of middleware systems.

3.2.2. Relation to this work

Several middleware properties are useful with respect to the development of the CAGISTrans framework. Middleware may help us to deal with many connectivity and interoperability problems. In this perspective, middleware is used as an enabling technology to provide the architecture and tools needed in the development of a CAGISTrans system. Our developed CAGISTrans architecture is built on and extended from TP-monitor functionality – supporting the specification and execution of transactions, database middle-

1. See http://www.transarc.com/Product/Txseries/Encina/Brochure2.0/encina.html.
2. See http://www.bea.com/products/tuxedo/index.shtml.
3. See http://www.oracle.co.uk/support/bulletins/net2.html.
4. See http://www.sybase.com/products/eaimiddleware/openclient.
5. See http://www.sybase.com/products/eaimiddleware/openserver.
6. See http://www.orbix.com/docs/orbix/orbix33.html.
7. See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp.

ware – supporting accesses to heterogeneous shared resource bases, and remote method invocation – enabling accesses and execution of remote actions.

Using the middleware principle, this work attempts to address some of the problems relating to the heterogeneous nature of cooperative environments. This includes the heterogeneity of resource management systems and the diversity of application areas. Further discussion of our specific use of middleware is provided in Chapters 6, 8, and 9.

3.3. Agent Technology

Agent technology – hereafter called agents – has along with the growth of the Web gained increasing attention from several research communities. Despite this, there is no commonly agreed definition of the term agent. A synthesis of definitions frequently used would be that an agent is a software entity that can autonomously perform delegated tasks, with or without communication with other entities, and by possibly moving from one host to another.

The following discusses the advantages of and challenges using agents in the cooperative work context.

3.3.1. Advantages of agents

The motivation for using agents in cooperative work support is mainly to address the distributed, heterogeneous and dynamic nature of cooperative work.

Agents as (semi-) autonomous entities – making them able to adapt to their environments – provide a suitable platform for the development of systems characterised by a high degree of distribution and openness. Agents are inherently distributed, thus enabling decentralised control. This, in turn, facilitates management of changes in specification and performance of cooperative tasks. Agents can also – thanks to their "social ability" – i.e., their ability to communicate and interact with other entities – handle sophisticated interactions with other partners. This feature, combined with their autonomous behaviour, makes it possible to provide high level support to users, both with respect to heterogeneity and the dynamic nature of cooperative environments.

Moreover, the mobility aspect of agents is useful when considering distributed environments. Mobility can be exploited mainly to reduce the communication cost (Harrison et al. 1995, Kiniry and Zimmerman 1997, Wong et al. 1999). First, because mobile agents can be transported from one host to another, the number of remote interactions can be reduced by bringing two entities that

frequently interact with each other to the same location. Second, agent mobility provides an efficient, asynchronous method for attaining information and services in rapidly evolving networks such as the Web. As a result, we can achieve a minimised amount of message interchanges and simplify the necessary distribution of computation.

Bradshaw (1997) uses agents to further simplify the complexity of distributed computing. The author suggests that this can be achieved by developing and providing agent-to-agent interfaces that are more sophisticated and uniform than the heterogeneous, traditional (possibly proprietary) program-to-program interfaces. From this perspective, the role of agents could be a means to overcome the limitations imposed by factors, such as the lack of interoperability among existing programs. In cooperative environments, this would allow the co-existence of several computing systems, each equipped with a generic agent interface, thus making it easier to perform distributed cooperative activities in heterogeneous environments (Bradshaw 1997). This also implies that the communication between cooperating systems would be performed at the agent level.

Another aspect is to exploit agents as a means to reduce the amount of interaction between humans and the computer system, thus allowing humans to concentrate on the actual work to be done rather than investing effort in doing "slave work" such as collecting, sorting and filtering information.

To summarise, the properties of agents such as autonomy, social ability, mobility, decentralised control and, to some extent, autonomous behaviour make agents well-suited for use in cooperative work. From a systems development point of view, exploiting these features may meet many of the challenges that arise due to the distributed, heterogeneous, and dynamic nature of cooperative environments.

3.3.2. Challenges with agents

Although the beneficial features of agents may be many, there are concerns that have to be considered.

Agents have been widely used for providing specific services such as message filtering and information retrieval. As yet, however, there are not many cooperative systems that fully exploit the autonomy of agents. The reason is probably that there are risks connected to the use of agents as they are utilised in Artificial Intelligence (AI). In particular, there is a chance that users might get the feeling of "losing" control. The more one builds intelligent support, the more one increases the risk that users will lose the whole picture of

what is going on. This is dangerous as the technology may still be too immature to fully deal with the dynamic nature of any working environment. Therefore, the trend is rather to use agents to assist humans with their work.

Further, we stress the necessity of providing support for handling interactions among agents. In particular, there is a need to provide a flexible mechanisms to manage and control the sharing of resources. This is based on the observation that agents mainly cooperate by means of sharing of resources and information. Thus, failing to address this may lead to conflicts and data inconsistency – i.e., loss of integrity.

In summary, the challenges of using agents in cooperative work mainly concern the way the properties of agents are to be exploited. Users that have highly autonomous and intelligent agents acting on their behalf might get a feeling of losing control. However, one of the key lessons learned in the AI community over the years is that even a little "intelligence" may, to some extent, help (Wooldridge and Jennings 1998). For example, even an agent keeping the user informed of which documents have been changed or updated would be potentially useful. Consequently, using highly "intelligent" agents in CSCW might pose problems, but implementing a certain degree of "intelligence" and autonomy with agents could be useful.

3.3.3. Agents and this work

The use of agents in this work concerns their mobility and cooperative aspects. Here, agents are primarily used as an implementation infrastructure. Further, due to the increased application of agents as user assistants in cooperative activities, we address the need of support to allow user agents to access necessary resources through a CAGISTrans system. Other reasons are illuminated in Chapter 10, concerning the integration into the CAGIS environment.

Existing agent system alternatives are many. The following surveys some of those relevant for this work.

- *Aglets* framework (Lange and Oshima 1998), initially developed by IBM Research Laboratory in Japan, and now maintained by the Open Source Community[1]. Essentially, aglets provide support for executing Java[2] programs and enable them to move in a network from one host to another host. An aglet can execute its code on a host, suspend this execution, and migrate to another remote host then start execution again. An aglet can communicate with another aglet, and in this way they may

1. See http://www.aglets.org.
2. Java is a trade mark of Sun Microsystems. See http://java.sun.com.

cooperate to achieve a goal. An aglet carries its state along with it when moving around, and it is able to move in the network following some specified path.

- *Voyager* (ObjectSpace Inc. 1997), developed by ObjectSpace as a product family consisting of an application server, an ORB, diverse service extensions (e.g. transaction service, etc.), predefined applets, and a set of algorithm libraries (so-called JGL Libraries). Voyager provides the support for mobile agents through the ORB, through which objects can travel from one host to another. It has an advanced messaging system that can track agents around the network, and forward messages onto them via secretary objects. It also supports remote invocation of agents and applications.

- *Mole agents* (Straßer et al. 1997), developed in the Mole project at the University of Stuttgart (Germany), similar to aglets. Mole provides policies for resource management, to allow charging for service and resource control. It differs from aglets in that it has a predefined resource management mechanism. In addition, mole supports an event service that can notify mobile agents.

- *Grasshopper* (Breugst et al. 1998), developed by IKV++, Germany to enable the user to create a number of applications based on agent technology. Grasshopper allows users to create agent applications enabling electronic commerce applications, dynamic information retrieval systems, advanced telecommunication services, and mobile computing systems. Grasshopper is implemented entirely in Java and based on CORBA. Thus, it allows a smooth integration of new technologies with legacy systems, and enables the use of both agents and client/server computing in one application.

The above overview illustrates the diversity of existing agent platforms. The choice of an implementation infrastructure for a CAGISTrans system was done based on ease of use, extensibility and modularity. In this respect, aglets were found the most appropriate platform. It provides us enough programming freedom without forcing us to follow specific design patterns, and at the same time allows us exploit its extensibility. Further discussion on the relevant features of aglets is provided in Section 9.1.

Chapter 4
State-of-the-Art Survey

4.1. Introduction

As initially pointed out, although traditional ACID (normally flat) transactions provide well-defined correctness criteria through *serialisability* (Bernstein et al. 1987) and efficient support for failure and exception handling through *recoverability* (Bernstein et al. 1987), they seem inappropriate for advanced applications (see Section 2.3). This has resulted in a call for more advanced and flexible transaction models. Since the early nineties, there has been an enormous effort in the attempts to develop such models. This chapter gives a review of some selected models. They were primarily chosen based on their relevance to this research, and to illustrate the diversity of existing solutions.

Existing transaction models can be divided into three main categories:

1. *Classical advanced transaction models*, primarily aimed at improving the ACID model to handle advanced applications, also called extended transaction models (ETMs).
2. *Newer transaction models*, particularly aimed at providing support for cooperative work, also called cooperative transaction models.
3. *Customisable transaction models*, especially aimed at tailoring transaction models to handle diverse situations.

These three categories will be covered in Sections 4.2, 4.3 and 4.4 respectively.

4.2. Classical advanced models

While database transactions have traditionally been mainly intended for business applications such as banking and the like, they have increasingly become relevant for more advanced applications such as CAD/CAM and design for manufacturing. In the course of the last decades or so, several extended transaction models have therefore been suggested and developed to improve the ACID model. The goal was to increase the application areas of transactions. Some of these are treated and reviewed in (Barghouti and Kaiser 1991, Elmagarmid 1992, Mohan 1994, Kaiser 1994). This section presents the models that have provided foundations towards unified support for advanced and cooperative work. These are *Nested transaction model* (Moss 1982), providing a basis for modular modelling of transactions, *Sagas* (Garcia-Molina and Salem 1987), providing a basis for transaction compensations, which were further exploited in *Open nested and multilevel transaction model* (Weikum and Schek 1992), a generalisation of Moss' nested transaction model, *Cooperative transaction hierarchy* (Nodine and Zdonik 1992), providing the idea of cooperative transactions using user-defined correctness criteria, and *Split and join transaction models* (Kaiser and Pu 1992), providing the idea of dynamic restructuring of transactions – outlined below.

4.2.1. Nested Transaction model

The nested transaction model was introduced by Moss (1982) to extend the flat transaction by splitting transactions hierarchically into several subtransactions (see Figure 4.1 for an illustration). A child transaction may start after its parent has started, and may commit locally. The committed local results are, however, released only when all of its parents up to the root have successfully terminated (Moss 1982). A parent transaction, on the other hand, may terminate only when all of its children have terminated, and it may execute another alternative transaction if a child fails. Note, however, that if this parent aborts, all of its children must also rollback, independently of their current state.

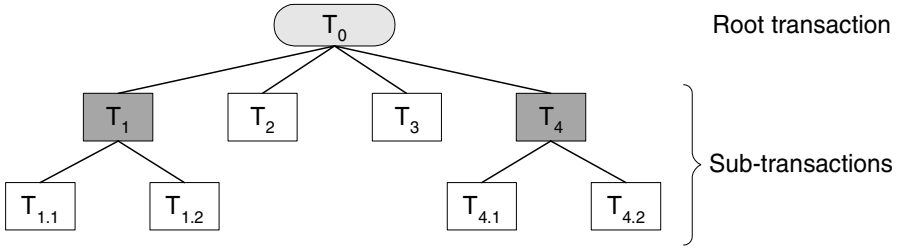


Figure 4.1 Illustration of the nested transaction model.

The nested transaction does not address cooperation since full isolation is required. However, it allows increased modularity, more concurrency and finer recovery granularity than the traditional flat transaction. Several sub-transactions can be executed simultaneously, and a subtransaction can abort without necessarily leading to the abortion of siblings.

4.2.2. Sagas

The Sagas model was introduced by Garcia-Molina and Salem (1987) primarily to deal with long transactions, by using the concept of compensating transactions. A Saga is a transaction that consists of *a sequence* of several ACID (sub-) transactions and associated compensating transactions. Each sub-transaction is allowed to commit individually. A compensating transaction is then used to explicitly undo its effect if the whole Saga transaction has to abort (Garcia-Molina and Salem 1987). Finally, a Saga can terminate only when all of its sub-transactions including any compensating sub-transactions have run successfully. This is illustrated in Figure 4.2.

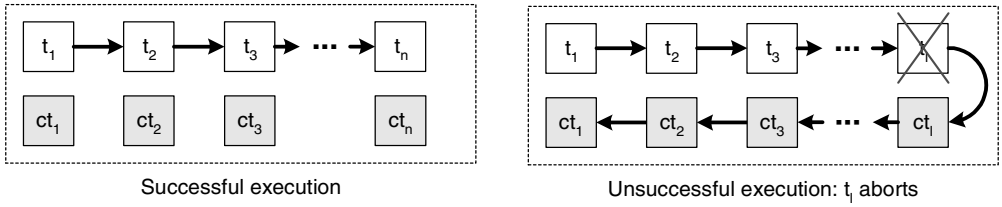


Figure 4.2 Illustration of execution of Saga transactions.

By allowing sub-transactions to commit, thus revealing their partial result(s), Sagas relax the full isolation requirement. This means that some degree of cooperation is permitted. However, the main drawback of the concept is that it is not fully implemented (Mohan 1994). This means that it is difficult to provide a complete evaluation of its performance and usefulness. Nevertheless, the idea has been used to implement other models. More specifically, the idea underlying compensating transactions has a lot of potential.

4.2.3. Open nested transaction and multilevel transaction models

Open nested and its specialisation multilevel transaction models were both proposed by Weikum and Schek (1992). They were suggested to improve the nested transaction model, by allowing sub-transactions to issue final commit and by using compensation as in Sagas to further enhance inter-transaction parallelism. As in the nested transaction model, they are both tree-based ap-

proaches, but the tree for the multilevel model is balanced (see Figure 4.3). This means that all leaf nodes in the tree are at the same level. Nodes in the multilevel transaction tree correspond to executions of specific operations at particular levels of abstraction. Ergo, all transaction trees have the same height (Weikum and Schek 1992).

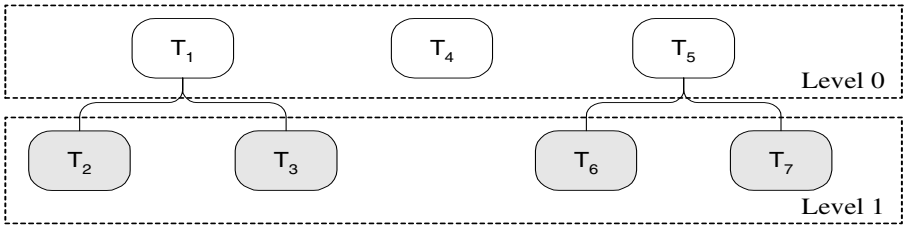


Figure 4.3 Illustration of the multi-level transaction model.

With both the open nested and the multilevel transaction models, all instantiated sub-transactions do not need to run successfully for the transaction to terminate. This means that the atomicity property is compromised, thus allowing flexibility. They also permit a higher degree of cooperation than the previous nested transaction model by relaxing isolation. Both models allow running transactions to reveal their partial results to other concurrent transactions. However, these partial results can be used by operations that commute only (Weikum and Schek 1992). This is managed by using semantic-based concurrency control (Weihl 1988). The main drawback of both models is that both stick to serialisability as their correctness criterion, making full flexible sharing of tentative data impossible. Finally, the implementation of the model is, to the present author's knowledge, still limited.

4.2.4. Cooperative Transaction Hierarchy

Cooperative transaction hierarchy is a transaction model proposed by Nodine and Zdonik (1992) for design environments. It is like the nested transaction model, a tree-based approach. In this case, the tree is organised into three main levels (see Figure 4.4): a *root*, one or more *transaction groups (TG)* and several *cooperative transactions (CT)*. The cooperative transactions correspond to the leaf nodes, which are grouped into transaction groups. They are each associated with a designer in the environment and can, within a transaction group, cooperate on some task. The cooperative transaction hierarchy model does not apply serialisability as the correctness criterion (Nodine and Zdonik 1992). Instead, it uses user-defined criteria. More specifically, these are *patterns* and *conflicts* – specified for each transaction group. Here, patterns are a set of rules for how operations can be interleaved, whereas conflicts are a set of rules that specify which operations are not allowed to run concurrently. Both

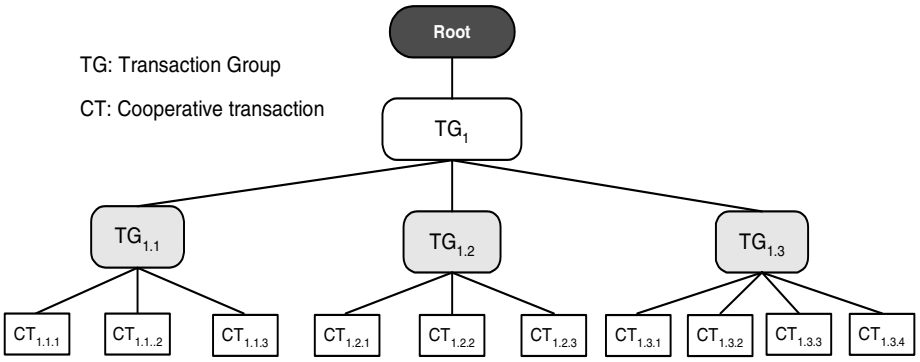


Figure 4.4 Illustration of the cooperative transaction hierarchies.

patterns and conflicts can be tailored to the needs of the application.

Although cooperative transaction hierarchy addresses cooperation, its main weakness is the need to define both patterns and conflicts in advance. Because of this, all sets of operations must be known a priori, which limits the use of the model to applications with a well-defined work structure.

4.2.5. Split and Join Transaction model

The split and join transaction model was proposed by Keiser and Pu (1992). Its main goal is to provide transactions with the ability to share resources by allowing dynamic restructuring of running transactions. It was originally developed for activities with uncertain duration, unpredictable developments, and interaction with other activities. The principle is basically to *split* a running transaction into two or more transactions and later *join* transactions by merging their resources (Kaiser and Pu 1992) – as illustrated in Figure 4.5.

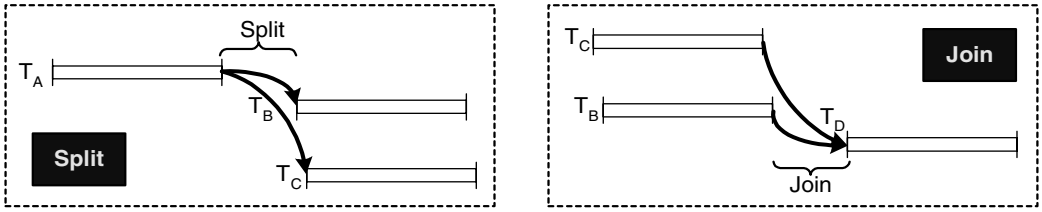


Figure 4.5 Illustration of the principle of split and joining transactions.

Thus, the model addresses cooperation among users by allowing the transfer of resources from one transaction to other transactions. Further, it introduces an adaptive recovery mechanism which allows part of the work done to be recoverable, and a committing transaction may release parts of its re-

sources. The main drawbacks are the emerging complex merging mechanisms. Moreover, the transactions resulting from the split command still have to obey a serialisability criterion. This implies that the two transactions still must be seen as two isolated transactions while running, thus prohibiting explicit cooperation among transactions.

4.3. Newer transaction models

The classical advanced transaction models do not cover all aspects of collaboration. In the past few years, researchers have therefore tried to develop transaction models and frameworks that further improve the support for cooperative work. This effort can be divided into two categories. The first category consists of developments based on cooperative work requirements, focusing on the CSCW perspective: *Coo* (Godart 1993, Godart et al. 1996), *EPOS* (Conradi et al. 1997) and *TransCoop/Coact* (de By et al. 1998, Wäsch 1999). The second category are those developed based on extensions of existing database fundamentals (e.g. locking protocols and timestamp ordering): *Relative Serialisability* (Agrawal et al. 1995) and *New Timestamp Ordering (Zhang et al. 1999)* - outlined below.

4.3.1. Coo

Coo is a research project aiming to develop a framework to support cooperation between software developers, by encapsulating software processes into long and cooperative transactions (Godart 1993). The transaction model was developed based on software development processes requirements. A new model with relaxed atomicity and relaxed isolation was suggested.

In Coo, relaxing the atomicity is achieved by allowing long transactions to save their intermediate results adopting the principles of partial rollback and savepoints from traditional transaction models (Fussell et al. 1981, Gray et al. 1981), thus minimizing losses in the case of crashes. Relaxed isolation allows several software processes to access these intermediate results without violating the correctness criterion.

Intermediate results are managed by applying three different object stability levels. These are *stable*, *semi-stable*, and *unstable*. An object is stable when it is fully consistent. This means that stable objects are those that all processes view as correct, specified in the correctness criteria. Semi-stable objects are those some specific processes may generate as tentative data, and can be seen as "consistent enough", but may violate some of the correctness criteria. For this reason, access to such objects is restricted to processes that satisfy specific semantic rules and integration constraints which are called safety constraints

and vivacity constraints in the literature (Godart 1993). Finally, unstable objects are those that do not satisfy the correctness criterion at all, and are currently locked by some processes. It is inaccessible until it becomes stable or semi-stable.

The three types of objects are stored in different databases, which are managed through the use of check-in/check-out operations – as illustrated in Figure 4.6. The double arrows show which databases contain which object types. As depicted in Figure 4.6, Coo provides two types of shared object bases in addition to private databases. The first base is a public object base which contains all stable objects. The second base is a scratch pad for software processes containing semi-stable objects. Data are exchanged between private and shared databases using the standard *check-in* and *check-out* operations. In addition, there are two other operations: *upward-commit* and *refresh*. The former is used by a process to make its intermediate results available to other processes in its scratch pad. The latter can be issued when these intermediate results have been modified.

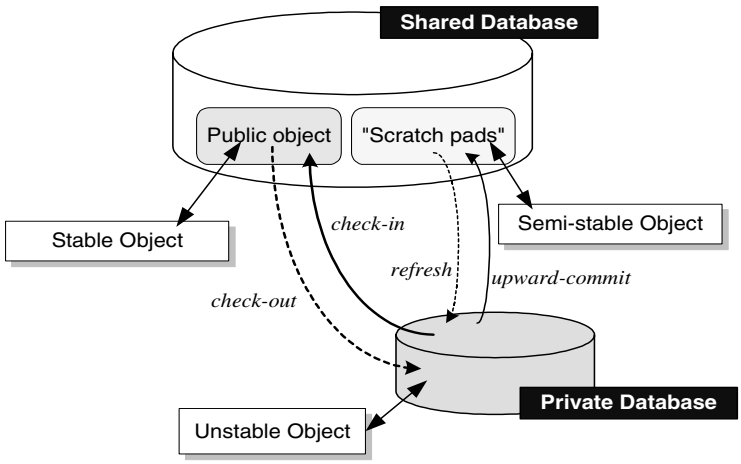


Figure 4.6 Relation between object consistencies and database types.

Coo introduces a formal framework. However, its use of perhaps too formal temporal logic may impose difficulties in practical implementations. Nevertheless, a working prototype is provided. Coo was specifically developed for software development processes. Correctness constraints and management of activities are tailor-made to suit such a type of cooperation. Moreover, the database system in Coo is developed specifically for this model. As a result, some aspects of cooperation, such as dynamic processes and heterogeneous resource bases or applications, may not be well supported. However, the idea of using three degrees of stability is useful on the way towards more flexible support for

cooperative work than with the classical advanced transaction models.

4.3.2. EPOS

EPOS (Conradi et al. 1997, Wang et al. 1998) was a research project at the Norwegian University of Science and Technology to develop a framework for quality assured software engineering. To support different software engineers, a database called EPOS-DB was designed to manage resources produced in the development process. For consistent access to the database, a flexible transaction model was proposed. This is another extension of the standard ACID transaction model to support cooperation between co-workers in a Software Engineering Environment (SEE). It shares some of the ideas of Coo in the sense that it is based on the use of workspaces (both private and common workspaces). The transaction model also uses check-in/check-out operation for interaction through/with the workspaces. As in Coo, Conradi et al. (1997) assumed that transactions for software development are generally long-lived. The use of transactions with nested structures, used in private and shared workspaces, was therefore suggested. In this view, the check-in/check-out model in EPOS also shares the basic ideas of Kim et al. (1984) and Bancilhon et al. (1985). However, EPOS has extended these by integrating the concepts with the software process technology and process modelling (Wang et al. 1998). Figure 4.7 illustrates this structure for a parent transaction T with N children T_i.

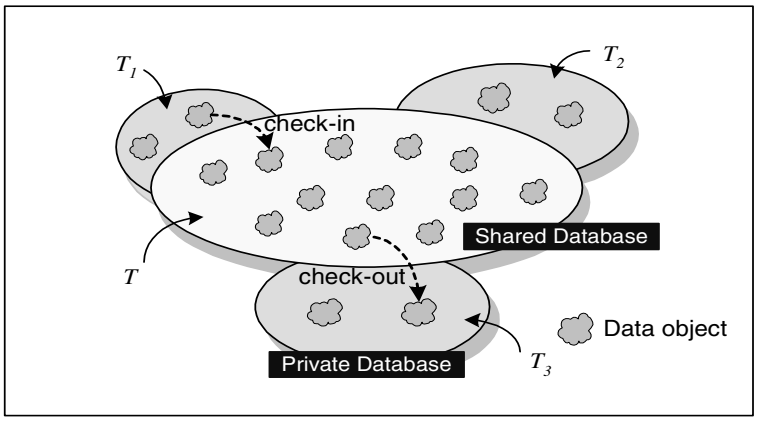


Figure 4.7 Illustration of a workspace with nesting structure in EPOS.

To deal with concurrency, EPOS uses locks to regulate access to a shared workspace. Awareness support is also provided to allow users to know about events that may affect their work. Correctness of execution is achieved by applying awareness mechanisms to help handling access conflicts. This means that notifications will be sent out to all involved parts to warn about potential conflicts. In this case, serialisability is not applied.

The *EPOS* transaction model shares many of the assumptions underlying Coo. It presents a useful framework that allows co-workers in an SEE to cooperate with a high degree of flexibility. The requirements for the transaction model were mainly derived from those applied in software development. Because of its flexibility, EPOS can be used to support other types of applications. However, the framework does not provide explicit guidelines for how this can be used effectively. This is left to the users to figure out. Nevertheless, a prototype implementing the basic ideas of EPOS has been developed.

4.3.3. TransCoop/CoAct

TransCoop was an ESPRIT basic research project involving research groups from GMD-IPSI (Germany), the University of Twente (The Netherlands) and VTT Information Technology (Finland) (de By et al. 1998, Wäsch 1999). The goal of the project was to develop a transaction model and a specification language to enable effective information sharing. In this presentation, we will consider the TransCoop transaction model only. The transaction model was called CoAct and was developed based on an extension of existing advanced transaction models. Their motivation was to overcome the limitations imposed by the use of a standard ACID model. The requirements for the transaction model were defined by using four application scenarios such as

- *cooperative authoring*, being based mainly on ad-hoc processes,
- *software engineering*, being characterised by semi-structured processes,
- *design for manufacturing*, being characterised by structured activities and
- *workflow*, focusing mainly on automated business processes.

Like other researchers in the field, they advocated the need for both relaxed atomicity and relaxed isolation. In addition, the model should allow users to explore several alternatives to solve a problem and to revise erroneous actions (retraction of decision). Moreover, it should provide support for management of alternative versions of data objects (private and shared data). Finally, the transaction model should allow the use of execution constraints to coordinate individual and joint work.

Basically, the ideas of CoAct were built on classical advanced transactions, such as those described in the previous section. These are *compensation and semantic-based concurrency control* from the open nested and multi-level transaction models, and *resource exchange* from the split and join transaction model. Further, they adopted the basic idea underlying *delegation* in ACTA (see Section 4.4.1). They then extended the Check-In/Check-out, versioning and

workspace models with sophisticated *history merging* mechanisms.

Figure 4.8 illustrates how different workspaces are used in CoAct. It is worth noting that, in CoAct, exchange of operations among transactions associated with specific workspaces is applied instead of explicit exchange of data among workspaces – e.g., as in Coo and EPOS. Correctness of interactions is checked by validating the history produced after each exchange (delegation, import or merging).

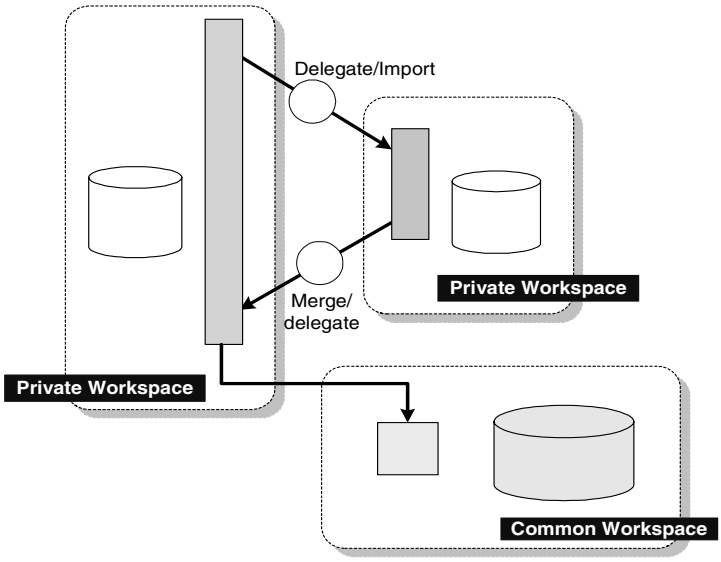


Figure 4.8 TransCoop workspaces and exchange operations.

TransCoop/CoAct is one of the models that has attempted to cover the broadest application area. They based their approach on four application scenarios that cover both structured and ad-hoc activities. Moreover, formal foundations based on provable mathematical formalismes are provided. This implies that the approach can be formally validated. However, the merging mechanisms introduced in the model may impose some complexity, which affects the overall manageability of the system. The available prototype proves its applicability in co-authoring applications.

4.3.4. Relative serialisability- RSR

Agrawal et al. (1995) suggested a new database approach to manage concurrent activities in collaborative environments. Their work was inspired both by the attempts by database researchers to relax the requirements of atomicity and isolation, and by the proposals by software engineering and design environments. Their main goal was therefore to develop a transaction model by merging flexible transaction models from collaborative environments and se-

mantic based correctness criteria.

The relative serialisability (RSR) model is built on the notion of *relative atomicity* originated from (Lynch 1983). Their idea is to specify relative atomicities of co-actions[1] to relax the traditional atomicity property – i.e., absolute atomicity. It determines how a co-action can be interleaved relative to other co-actions without breaking the pre-defined consistency requirement for the actual collaborative activity (Agrawal et al. 1995). So, before a collaborative activity takes place, a *collaboration channel* must be established. Then, by connecting to this channel, different transactions may cooperate on the same data objects following the relative atomicity specifications. Correctness of execution is checked against the so-called *relative serialisability (RSR)* correctness criterion, a more relaxed criterion than traditional (conflict) serialisability (SR). Thus, the underlying assumption is that any execution obeying the RSR criterion would preserve the consistency of the database even if it is not serialisable.

Figure 4.9 illustrates how different transactions can specify their atomicity relative to other transactions, and how this specification can be used to verify the relative serialisable executions of concurrent transactions. Here, the

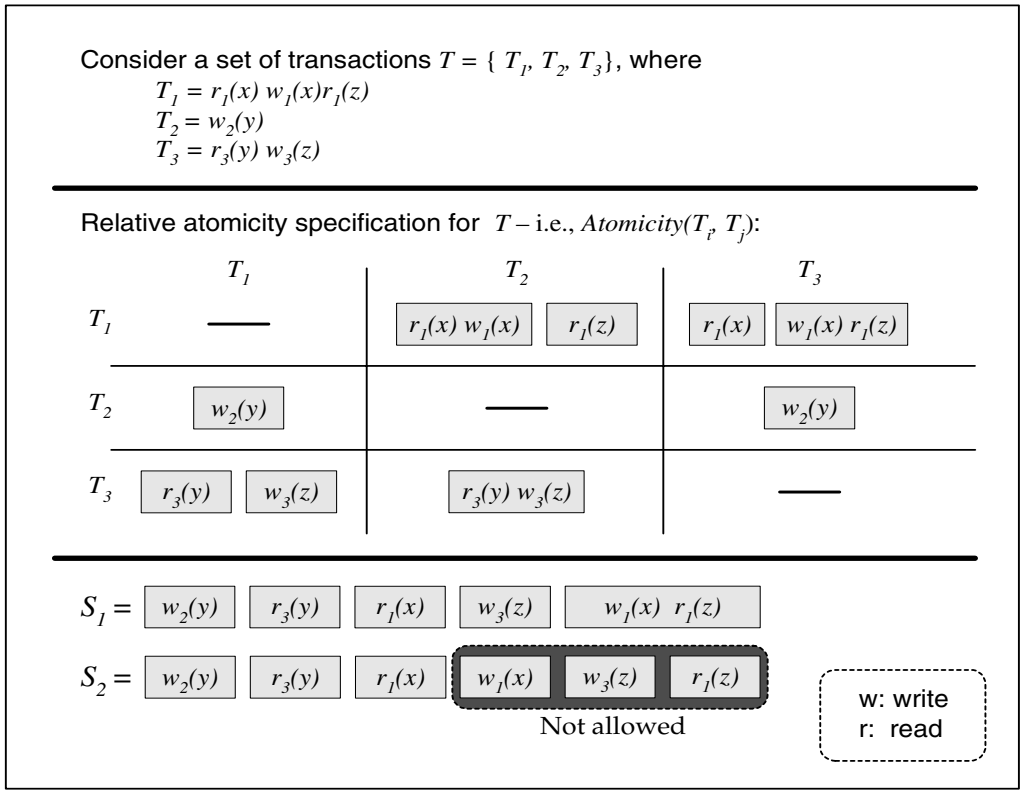


Figure 4.9 Illustration of the use of the relative serialisability criterion.

1. A co-action is a sequence of read and write operations executed on data objects.

relative atomicity is denoted by *Atomicity*(T_i, T_j). It is defined based on the notion of *atomic unit*. An atomic unit of a transaction T_i relative to another transaction T_j is a sequence of T_i's operations within which no operations of T_j are allowed to be executed. This means that *Atomicity*(T_i, T_j) consists of all atomic units of T_i relative to T_j.

In our example, we consider a set T consisting of three transactions T_1, T_2, and T_3. As depicted in Figure 4.9, the relative atomicity relation *Atomicity*(T_1, T_2)=®[$r_1(x)w_1(x)$], [$r_1(z)$]¯ means that T_2's operations may appear between [$r_1(x)w_1(x)$] and [$r_1(z)$], but must not be interleaved within [$r_1(x)w_1(x)$]. *Atomicity*(T_3, T_1)=®[$r_3(y)$],[$w_3(y)$]¯ means that T_1's operations may appear between [$r_3(y)$] and [$w_3(y)$], before [$r_3(y)$] and after [$w_3(y)$]. With respect to the relative atomicity specifications in Figure 4.9, execution S_1 is correct since it does not contain any illegal interleavings. By contrast, according to *Atomicity*(T_1, T_2) the three last operations in S_2 are illegal. Ergo, S_2 is not correct.

To guarantee the relative serialisable execution of co-actions, a new lock protocol is suggested. It extends the standard 2-phase lock protocol (2PL) (Bernstein et al. 1987) by introducing the notions of *push-forward* and *push-backward* locks. To handle conflicts, both such locks have to be acquired before an involving operation sets a normal lock. A push-forward lock causes any conflicting operation to be delayed until the last operation of the actual atomic unit with which it conflicts is run, whereas a pull-backward lock is used to move operations backward before the start of an atomic unit. Acquiring push-forward and pull backward locks is only possible if the cause of pulling or pushing operations does not change their effects.

The relative serialisability model is based on formal foundations. It extends traditional databases with a more flexible correctness criterion to support collaboration. A typical application area is design environments. However, to be able to specify relative atomicity, one must know the complete sets of operations before the involved transactions can be executed. This implies that dynamic applications are not well supported because new operations may not be introduced once transactions are in progress – i.e., transactions may not be adjusted dynamically. Also, from the illustration above, we can see that the "cross product" of semantic information must be provided. This means that each transaction that wishes to relax atomicity must first analyse any other potentially existing transactions in order to create atomicity specifications. Although the algorithm for guaranteeing correctness here does not necessarily create any major difficulties, the space complexity of creating the atomicity specifications could put a severe limitation on the usability of the RSR approach.

4.3.5. New timestamp ordering

Zhang et al. (1999) suggest a new timestamp ordering (NTO) approach that allows both traditional short transactions and long cooperative transactions to be run within the same system. Their method is in the same category as that of Agrawal et al. (see Section 4.3.4) in the sense that they also use database concepts as a starting point for their approach.

The main idea with this new timestamp ordering is to facilitate the co-existence of long, cooperative transactions and traditional, short transactions. Fundamental to the NTO model is that conflicts between operations of cooperative transactions and traditional (ACID) transactions should not crash the cooperative ones. This means that although two operations – like a *read* operation of a cooperative transaction and a *write* operation of a traditional transaction – conflict, the cooperative transaction will not need to abort (see examples below). Rather, it is assigned a new *virtual* timestamp, allowing it to incorporate the recent updates into its own processing, and both transactions may proceed as normal. By enabling this, it has been shown that by still using the traditional timestamp ordering (Bernstein et al. 1987) on all involved short transactions, the serialisability among these short transactions can be preserved. However, for the long cooperative transactions, the NTO scheme puts high priority on the last read or write conflicts in developing the correctness criteria. Although this implies that these transactions may not satisfy the serialisability criterion, correctness of the final result can still be achieved, by fulfilling a relaxed serialisability criterion called final conflict serialisability (Zhang et al. 1999).

Figure 4.10 illustrates how a long cooperative transaction T is run together with several short transactions, and how the suggested timestamp ordering approach handles conflicts. Let $t(T)$ be the timestamp of the transaction T. The following are examples of the different situations that may occur:

(1) T reads a value A – i.e., $r(A)$ – that later is updated by T_1 – i.e., $w_1(A)$ – before it commits. This means that $r(A) < w_1(A)$. T_1 has started before T and therefore it has a smaller timestamp than T – i.e., $t(T_1)<t(T)$. With traditional TO, T_1 would have to abort and restart with a larger timestamp. Instead, the scheduler partially rolls back T and lets it re-run the read operation $r(A)$. No new timestamp is needed since $t(T_1)$ is already smaller than $t(T)$.

(2) T tries to read a value B written by T_2 after T_2 has committed – i.e., we now have $w_2(B) < r(B)$. Since $t(T) <t(T_2)$, traditionally we should abort

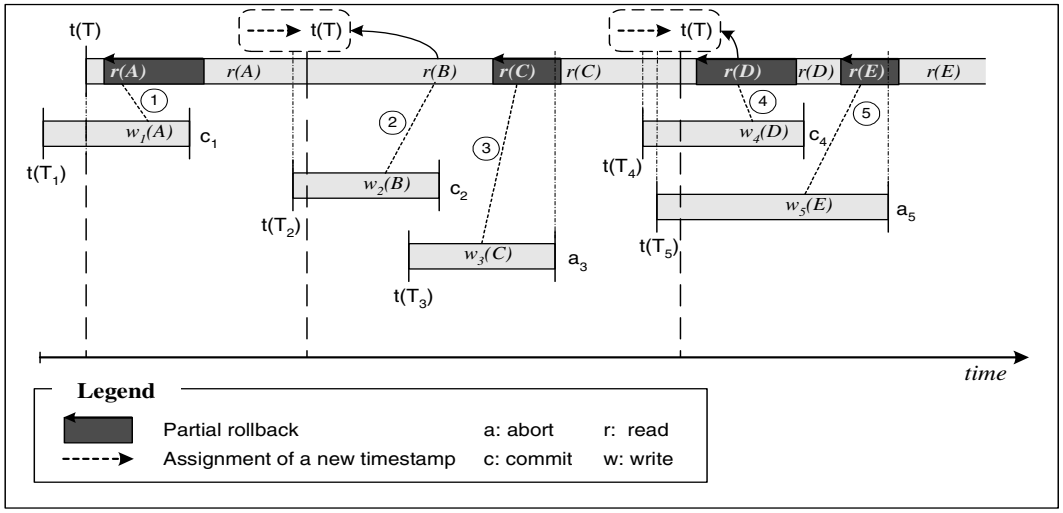


Figure 4.10 Illustration of the use of the new timestamp ordering approach.

T. Instead, *T* is assigned a new timestamp larger than T_2 and *T* may proceed.

(3) Now, T_3 updates *C*, which *T* reads before T_3 aborts – i.e., we now have $w_3(C)\ r(C)\ a_3$. If T_3 had committed, we would have to assign T a new timestamp as in (2) before it could proceed. However, since T_3 aborts, *r*(*C*) is "a dirty read" and the scheduler has to partially roll back *T* so that the read can be re-run. In this case, due to T_3's abort, no new timestamp is needed.

(4) *T* has read a value that T_4 later updates and then commits – i.e., we now have $r(C) < w_4(C)$. Since the newest timestamp for *T* is still smaller than for T_4 – i.e., $t(T) < t(T_4)$, *T* must incorporate T_4's changes as in (1) and acquire a new timestamp as in (2). Note, if $t(T_4) < t(T)$, this case would be the same as (1).

(5) We have a conflict between *r*(*E*) and $w_5(E)$, and this occurs before T_5 aborts – i.e., we now have $w_5(E)\ r(E)\ a_5$. Here, since T_5 aborts, *r*(*E*) becomes "a dirty read". Therefore, *T* has to partially roll back, and re-run *r*(*E*). Note, because of the new timestamp of *T* from (4), we now have $t(T_5) < t(T)$. If not, this case would be similar to (3).

Following the final conflict serialisability principle with the NTO scheme, these five cases cover all conflict situations that may occur – in addition to those covered by traditional timestamp ordering (TO), and the corresponding

TO rules produce correct transaction executions. To prove correctness, the final conflict serialisability is applied. Executions of short transactions will still follow the standard basic TO rules. Hence, they are serialisable (Bernstein et al. 1987). F-conflict serialisability is only applied to executions of cooperative transactions with respect to short transactions. Note that because of their explicit focus on final conflicts, Zhang et al. (1999) do not seem to put much emphasis on the impact of the timestamp changes on previously executed operations.

The new timestamp ordering approach is a useful approach, which bases the support of cooperative transactions on extensions of existing database concurrency control mechanisms. Support for dynamic activities is better than for the relative serialisability method (see Section 4.3.4) since a definition of operation sets is not needed before execution.

4.4. Customisable transaction models

The effort to develop cooperative transaction models has contributed to an increased acceptance of transaction models in other communities than the database community, such as CSCW. However, it is a widely accepted fact that there are problems that remain unsolved. For instance, existing models have restricted application areas. Therefore, the trend is rather towards the development of frameworks for transaction models, that can be tailored to different situations.

4.4.1. ACTA

ACTA[1] (Chrysanthis and Ramamritham 1990, Chrysanthis and Ramamritham 1994), is a transactional framework that allows formal reasoning and synthesising about the properties of transaction models. It is a formal framework that can be used to specify transaction models by determining the effects of transactions on other transactions (interaction between transactions) and the effects of transactions on objects. The building blocks of ACTA are illustrated in Figure 4.11.

- *The effects of transactions on other transactions* are specified or determined by *inter-transaction dependencies* – i.e., dependencies among transaction arising from structure (e.g. nested transactions) or from behaviour (e.g. reads/writes of transactions).

- *The effects of transactions on objects* are determined by:

1. This name was adopted from the latin word “acta”, meaning action.

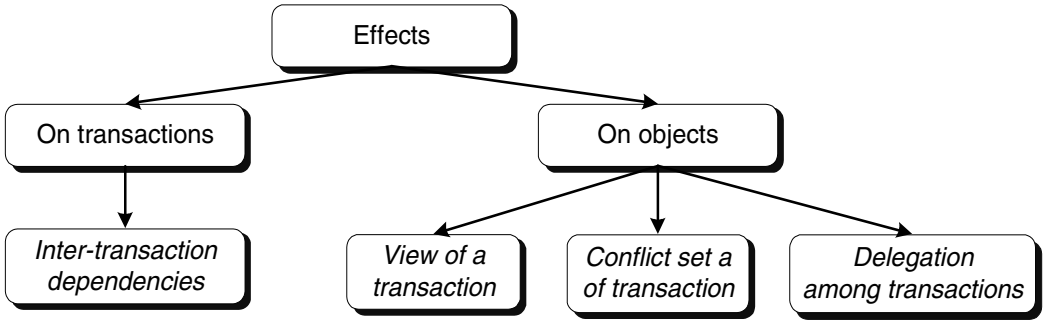


Figure 4.11 Effects on transactions and objects in ACTA (Chrysanthis and Ramamritham 1994).

- *view of a transaction,* specifying the objects and the state of these objects visible to a transaction at a specific point of time
- *conflict set of a transaction*, containing a set operations that are in progress and that a transaction must use to determine conflicts
- *delegation among transactions*, allowing a transaction to delegate the responsibility for committing or aborting a specific set of operations to other transactions

ACTA uses first-order logic to capture properties of transactions, such as visibility, consistency, recovery, and permanence. A new extended transaction model can be specified by using *ACTA-axioms*. Based on the above main building blocks, these can be used to "invent" new transaction models or just extend existing models by methodically modifying their definitions. Using ACTA a Saga can, for instance, be extended by giving it a new nested structure (Chrysanthis and Ramamritham 1992). In view of this, the usefulness of the ACTA framework depends on which models we want to specify and modify. Its power lies in the ability to represent structural behaviour of transactions and the dependencies between them. However, although ACTA is a very useful formal tool to specify and verify new transaction models, it is merely a formal and theoretical framework, providing no tools or mechanisms to explicitly allow transactions to be operational during runtime.

4.4.2. ASSET

ASSET – A System Supporting Extended Transactions – is a transactional framework suggested by Biliris et al. (1994) to implement some of the ideas of ACTA with O++ language primitives. The idea is to allow coding of extended transaction models (ETMs) using the O++ programming language, and to make these models operative on top of a DBMS. The ASSET primitives are `in-`

`itiate(func, args)` – registering and initiating a new transaction executing `func` with arguments `args`, `begin(t)` – starting a transaction execution with a transaction id *t*, `abort(t)` – aborting a transaction *t*, `commit(t)` – committing a transaction *t*, `wait(t)` – waiting for a transaction *t* to terminate, `self()` – returning the id of the currently executing transaction, and parent() – returning the id of the parent of the current transaction. In addition, ASSET introduced new primitives allowing a transaction model designer to delegate resources among transactions and permitting conflicting operations – thus allowing co-operation among them. These are (see Figure 4.12):

- `delegate(t_i, t_j, obj_set)`, which tells transaction t_i to transfer its responsibilities for the operations it runs on `obj_set` to t_j.
- `permit(t_i, t_j, obj_set)`, which allows t_j to perform operations upon obj_set that normally conflict with t_i's operations.
- `form_dependency(type, t_i, t_j)`, which defines a dependency type between t_i and t_j.

In (Biliris et al. 1994), the authors use these primitives to specify ETMs, demonstrating the usefulness of the framework. The strength of the framework is its ability to capture the characteristics of advanced transactions and use this to implement and execute these on a database system. However, the ASSET's low level focus is risky with respect to programming bugs. Moreover, since a transaction model has to be coded and compiled before the involved transactions are executed, there is a need for complete a priori knowledge about the tasks to be carried out as well as the allowed sharing "patterns". This makes it impossible to provide adequate support for dynamically evolving collaborative activities. A change in a model would require full interruption and redesign. Regarding heterogeneity support, ASSET was primarily designed for

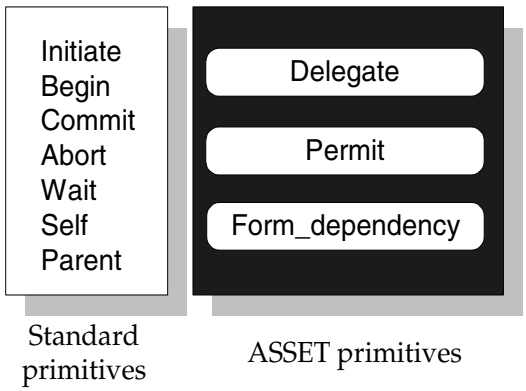


Figure 4.12 The ASSET primitives.

databases supporting the O++ programming language. Therefore, only systems supporting this language would be able to directly incorporate the primitives of this framework.

4.4.3. TSME

TSME – Transaction Specification and Management Environment (Georgakopoulos et al. 1994, Georgakopoulos et al. 1996) is another transactional framework, allowing specification and implementation of ETMs. TSME was developed as a complete transaction management system with a programmable transaction manager that enforces the specified transaction models during runtime. The main building blocks of transaction specification in TSME are dependencies. These are classified into *state dependencies*, specifying dependencies related to transaction states; `begin`, `abort` and `commit`, and *correctness dependencies*, specifying dependencies related to correctness criteria; determining which concurrent executions of complex transactions preserve consistency and produce correct results.

Figure 4.13 shows the block diagram of the TSME system architecture. The TSME system consists of two main components:

(1) A *transaction dependency specification facility (TDSF),* providing an environment for specification of complex transactions, specification of

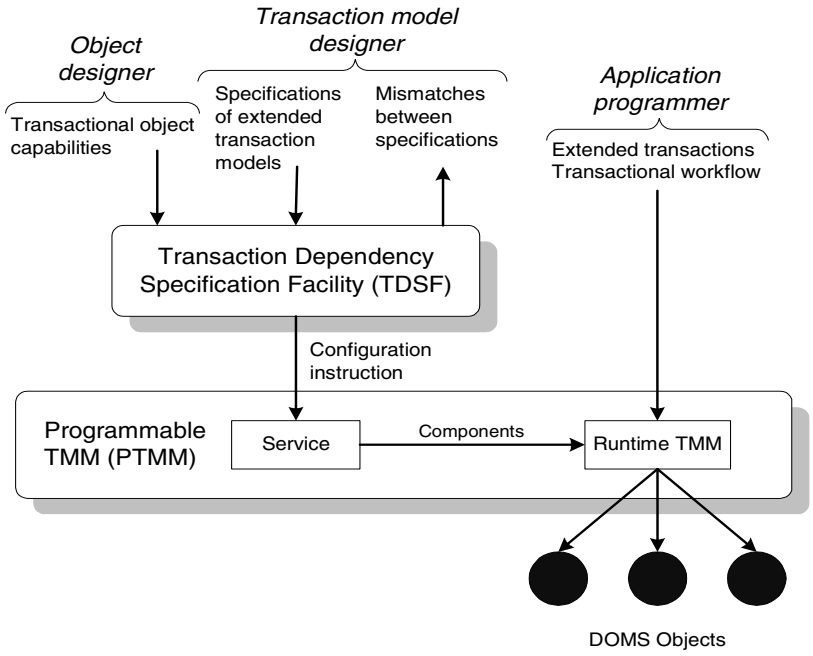


Figure 4.13 The TSME system architecture (Georgakopoulos et al. 1996).

transactional objects, and control and management of the specifications – e.g., catching mismatches between specifications.

(2) A *programmable transaction management mechanism (PTMM)*, providing services to enforce the specified ETMs at runtime.

The aforementioned dependencies are submitted by a transaction model designer to the TDSF which translates them into combinations of event-condition-action (ECA)[1] rule definitions and commands, and sends them to the PTMM that will manage and control the execution structure of the individual transactions. Once processed, the specifications of extended transactions are stored in a repository managed by the TDSF. The PTMM component makes sure that the specified dependencies are preserved at runtime.

TSME is a promising framework that has further demonstrated the advantages of allowing a model designer to specify several application specific transaction models within a single environment, and supporting them at runtime. One of the main strengths of TSME is its extensive support for execution control, allowing sophisticated coordination of transaction execution. However, TSME does not provide support for dynamic restructuring, making it less appropriate for dynamically evolving environments. Moreover, the transaction manager component in TSME appears to be built from scratch, and integration with and support for other resource bases than DBMSs, such as Web servers, is to the present author's knowledge de-emphasised.

4.4.4. RTF

RTF – Reflective Transaction Framework (Barga and Pu 1997, Barga 1999) is yet another framework, with the aim at specifying and implementing application specific ETMs. The main focus of RTF was to develop modules that implement existing ETMs on top of commercial TP-monitors. The base components are *transaction adapters* – i.e., add-on software modules providing extensible transactional services for advanced applications (see Figure 4.14). Using these adapters, RTF extends the facilities of a TP-monitor, allowing it to execute transactions beyond the ACID models.

Initially, RTF was implemented on top of Encina[2], demonstrating the applicability of the framework. It is important to note, however, that RTF does not provide support for user-defined correctness criteria. This was beyond the scope of RTF, and was left for future studies (Barga 1999). Rather, the focus was

1. The ECA concept was introduced in (Dayal et al. 1990).
2. See http://www.transarc.com/Product/Txseries/Encina/Brochure2.0/encina.html.

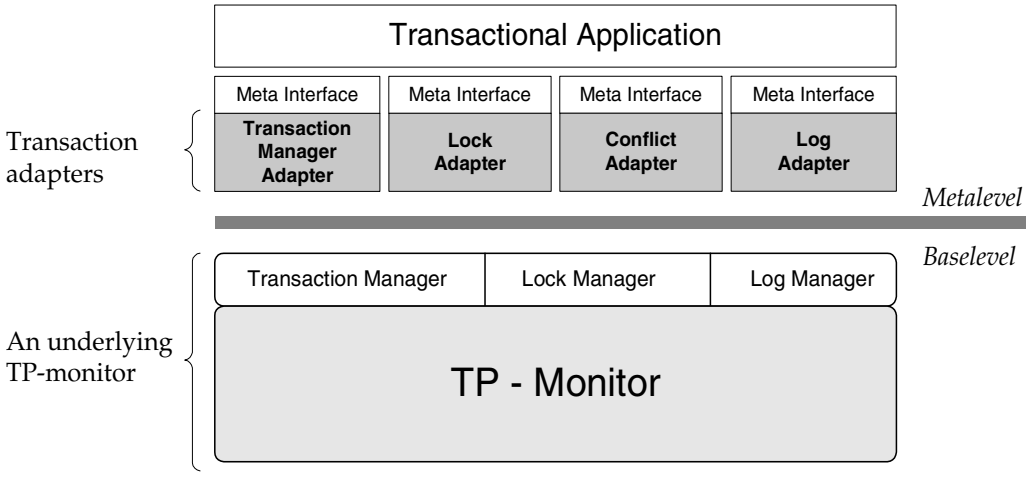


Figure 4.14 The transaction adapters in RTF (Barga and Pu 1997).

on provision of extensible lock protocols handling concurrency. Moreover, RTF is mainly a database-centred framework, and does not explicitly address the support for other resource bases that are not directly supported by the actual underlying TP-monitors. Hence, support for heterogeneous resource bases is de-emphasised.

Part II

DESIGN AND ARCHITECTURE

URN:NBN:no-2103

URN:NBN:no-2103

Chapter 5
Requirement Analysis

In order to be able to reason about requirements for cooperative work, we must know the characteristics of this type of work. Consequently, this chapter starts by describing a motivating scenario in Section 5.1. In Section 5.2, we analyse the characteristics of cooperative work, and use this to outline the requirements for cooperative work. This chapter then discusses cases from the motivating scenario that can be related to the requirements.

5.1. Motivating scenario

In the following we consider a case[1] of a modern software engineering process to illuminate the characteristics of cooperative work and the requirements that it imposes. We will base our examples in Chapter 6 and Chapter 7 on this scenario.

Aviasoft Inc.[2] is a software company that specialises in software for aviation. At present, the company is developing a computer-based flight instrumentation system (CFIS). The main purpose of this software to calibrate landing systems at airports.

Aviasoft is an international company with divisions in the USA, Canada, India, and South Africa as well as Norway (see Figure 5.1). Its headquarters are in Trondheim. Each division is connected to headquarters through the Internet. Documents produced in the development process are stored where they are generated, but they ought to be accessible from other divisions. In this way,

1. This case was inspired by a software development project that the present author was a member of, for a Norwegian aviation software company.
2. Aviasoft is a fictive name used here for illustration purpose only.

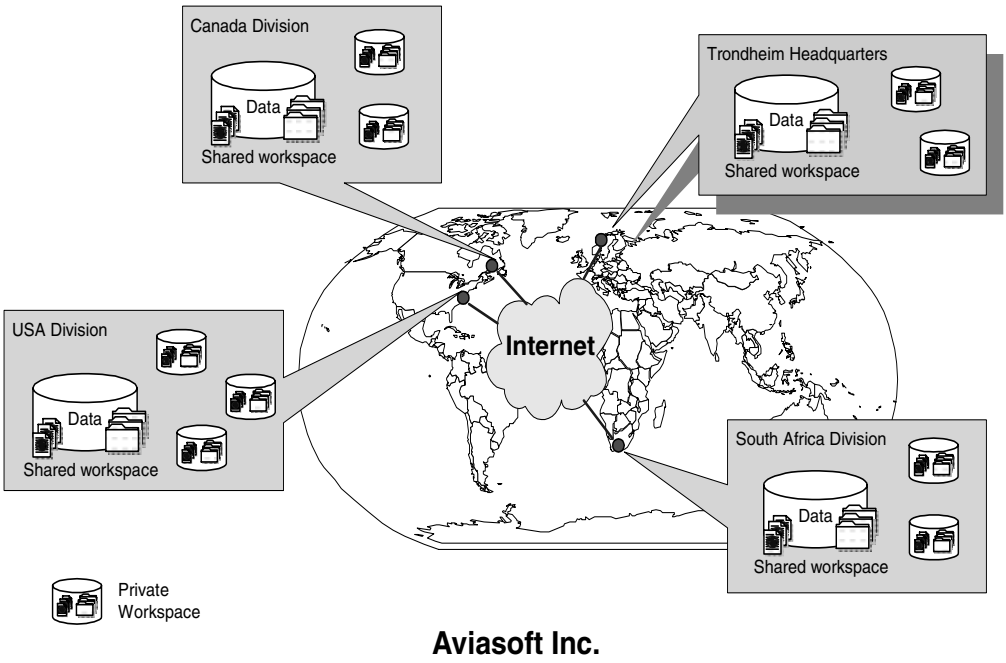


Figure 5.1 Illustration of distribution and organisation of Aviasioft.

they can be updated and retrieved as needed. The type of documents are development agenda[1] (mainly XML-based forms describing the development process each engineer must follow, and project state documentation), source code (written in Java), code documentation (written in HTML and Latex format, describing the functionality and the use of a module), and several specification documents (written in HTML and Latex format – such as requirement specification, design specifications, quality assurance specification, etc.).

The CFIS software consists of three main modules:

- Interface to plane sensors – called the *Interface* module – i.e., a driver module for plane sensors
- Graphical user interface consisting of menus, flight instruments, etc. – called the *GUI* module – i.e., the interface to the user
- Processing module being a main module – called the *Process* module – i.e., a module collecting data read by the Interface module, processing these and converting them to internal representations

Each module is developed by groups of 3 to 20 engineers in the divisions located in the USA, Canada, India, South Africa and Norway. They are put together into a single software artefact at a later project stage.

1. A development agenda is the same as a development plan.

The development process goes through several phases before the software is installed in a plane. These consist of analysis, requirements specification, and design, coding, and testing. Further, to meet the strict reliability requirements for aviation software, Aviasoft applies independent code inspection to discover bugs and software faults. This is often performed while the involved engineers are programming their modules. To ensure objective results, code inspections are normally performed by one or more engineers from other divisions than that of the programmer.

5.2. Cooperative work characteristics

5.2.1. Definition of cooperative work

The term cooperative work may have different meanings within different disciplines – sociology, computer science, etc. Therefore, it has no widely accepted exact definition, and several attempts can be found in the literature. One of the earliest definitions was given by (Marx 1867) as:

> "*When people work together side by side in accordance with a plan, whether in the same process, or in different but connected processes, this form of labour is called cooperation.*"

This is not a complete definition. It is too narrow since it assumes that cooperative work is always planned. For example, consider our Aviasoft scenario. A requirement specification must be in place before the coding process starts. However, during coding, the specification may have to be adjusted due to some inaccuracy or imposed limitations. Therefore, the involved engineers may be required to perform different "unplanned" actions.

In conclusion, the assumption that cooperative work is always planned is not accurate. An intuitive and generic definition based on the intention with the work to be performed could be: "a work process involving several people *acting together*, in a *shared context*, performing some tasks in order to achieve a *pre-specified common goal*". This fits well with the context of Aviasoft.

5.2.2. Characteristics of cooperative work

Several researchers from the CSCW community have done analyses of cooperative work characteristics (Rodden and Blair 1991, Schmidt and Bannon 1992, Schmidt and Rodden 1996). Using these analyses, our Aviasoft example and the definition above as starting points, the following are various dimensions that are relevant for the provided transactional support (see also Table 5.1 for a summary).

Dimensions	Comments
Degree of cooperation	Any cooperation spans from separate independent tasks to a tight collaboration.
Group size	The participants may be single users, or they may be organised in teams, each consisting of several parties.
Temporal distribution	The interaction among the involved parts may be synchronous – i.e., real-time interaction, or asynchronous – i.e., deferred interaction.
Geographical distribution	The participants may be co-located or dispersed.
Work duration	The duration of any interaction is unpredictable, thus being impossible to estimate in advance.
Work structure	The activities range from being well-structured – e.g., workflow, via semi-structured – e.g., software development, to ad-hoc – e.g., collaborative learning.
Resource usage	The artefacts used and their access properties may vary over time – i.e., with variable degrees of resource sharing.

Table 5.1 Dimensions of cooperative work.

(1) The *degree of cooperation* among the involved parts may differ depending on the situation needs. Sometimes a tight cooperation or collaboration may be required in order to get the work done, while in other cases, cooperation may not be required at all. The means that the spectrum of cooperation spans from individual separate tasks to collaborative activities.

(2) The *size of the group* of parties engaged in a work process is not stable, and may vary over time. This means that the participants may be single users, or they may consist of several individuals organised in teams.

(3) *Temporal distribution* is quite usual. The interaction among the participants may be synchronous – i.e. real time interaction, or it may be asynchronous – i.e., deferred interaction.

(4) *Geographical distribution* is normal. A cooperative activity may take place in the same location, thus all participants are co-located, or it may be carried out at different places, thus one or more participants may be dispersed.

(5) The *work duration* is not always possible to predict in advance. Sometimes, a complete set of actions can be specified before the task is initi-

ated. For such an activity, it is possible to estimate an approximate duration. For other cases, such an estimation is neither practical nor feasible.

(6) The *work structure* may differ depending on the type of activities. Cooperative activities range from being well structured – e.g. activity that involves the use of workflow, via semi-structured – e.g. software development – to ad-hoc – e.g. publication activities.

(7) The *resource usage* is not predictable. The artefacts used in a cooperative activity and their access properties may vary from time to time. This means that the degree of data sharing may not be stable.

5.3. Requirements for transactions

A conclusion that we may draw from this is that cooperative work is inherently diverse and dynamic. This makes it important – though challenging – to provide unified support. Using the cooperative work characteristics above, we have set up a set of requirements that a successful transaction model should satisfy (Ramampiaro and Nygård 1999). A summary is shown in Table 5.2.

5.3.1. Transactional properties

Due to their rigidity, the necessity of relaxing the ACID properties – thus allowing tighter cooperation – has traditionally been advocated. However, relaxation alone may still not be sufficient. Most people agree that consistency and durability should still be preserved in order to provide correct and reliable data management. In addition, there is a need to provide tailorable atomicity – i.e., finer abortion scheme – and controlled sharing of intermediate results – i.e., an adjustable isolation requirement.

TR1 *Tailorable atomicity.* First, to prevent long transactions from wasting work upon failures or exceptions, the transaction model should provide a fine grained abort management scheme, allowing *partial rollback* in addition to automatic total rollback. This implies that it should be possible to customise the (failure) atomicity requirement to fit different situations.

TR2 *Controlled sharing of intermediate results.* Since cooperation among parties may be necessary, enabling resource sharing is crucial. For this reason, requiring transactions to always execute as isolated units of work is unacceptable. This implies that a transaction model should allow flexible interleaving between transactions when this is

Requirements	Description
TR1 Tailorable atomicity	The transaction model should allow *partial aborts* in addition to traditional total aborts.
TR2 Controlled sharing of intermediate results	The transaction model should allow the isolation properties to be relaxed to enable cooperation, while still providing adequate concurrency mechanisms to prevent inconsistencies.
TR3 Support for open-ended execution	The transaction model should support runtime delegation of responsibility and interactive executions of transaction to cope with unpredictably.
TR4 Distribution and heterogeneity	The transaction model should meet the distributed and diverse nature of cooperative work.
TR5 Awareness	The transaction model should provide notification mechanisms to make participants aware of events that may affect their actions and help them handling potential conflicts.
TR6 Temporal data management	The transaction model should provide participants with enough knowledge about activities performed and changes made on shared resources to allow these participants to leave and join an activity without facing too severe restrictions.
TR7 Access control	The transaction model should provide rules defining which objects can be accessed by whom.

Table 5.2 Requirements with transaction models.

needed. However, concurrency control mechanisms are still necessary to prevent inconsistencies. In conclusion, the isolation property should be customisable to fit different sharing needs.

5.3.2. Transactional behaviour and support

The nature of cooperative work imposes several additional requirements that must be taken into account. These include those related to transactional behaviour and support. Transactional behaviour is associated with how transactions are executed at runtime. This includes execution duration, interactiveness and delegation of resources and responsibility. Transactional support is associated with how transactions address the distribution and heterogeneity of cooperative work environments.

TR3 *Support for open-ended execution*. The fact that we may not be able to predict the composition of actors involved in a cooperation, and that the interaction may have an uncertain duration and form, implies that the transaction model should enable open-ended execution of transactions. This means that it should be possible for a transaction to delegate resources to other transactions at runtime. This could, for instance, allow a transaction to make its resources available to other transactions although it aborts. Further, due to unpredictably, the ability for users to start and stop their transactions as desired is useful, making interactive support necessary.

TR4 *Support for distribution and heterogeneity*. Cooperative work is diverse in nature. Distributed and heterogeneous environments are commonplace. Therefore, it is important that the transactional model considers these two issues.

5.3.3. Services provided

To provide flexible transactional support, there are several services that should be provided to address the needs of cooperative activities. These include awareness – which is necessary as a result of allowing sharing. Then there is temporal data management – which is necessary to help cooperating parties to manage their tasks. Finally, there is access control – which is necessary for security and privacy reasons.

TR5 *Awareness*. Flexible sharing of information calls for mechanisms that allow involved actors to be aware of events that may affect their actions. This include those related to access to shared objects. This means that with relaxed isolation, awareness services should be provided to help handle potential conflicts.

TR6 *Temporal data management*. It is desirable to allow members of a cooperative activity to leave and join their group without facing undue restrictions. For this reason, the transaction model should provide these members with enough knowledge about activities performed, and about changes made on shared resources while they were away.

TR7 *Access control*. Sharing of artefacts is important in cooperative settings. Still, users may not want to share all of their objects. As a result, there is a need for a set of rules that define which data/ objects can be accessed by whom.

5.4. The requirements and the scenario

Recalling our Aviasoft illustration and considering the following cases, we can illustrate the necessity of meeting the above requirements. Specific aspects of these requirements will be further addressed throughout the thesis.

Case 1: Document production

It is common to require that documents and program codes are inaccessible to others until they reach a mature phase – e.g., when they are released. However, after such a phase one may still want to update released documents so as to satisfy some new requirements, or for bug fixing. At the same time, other engineers may want to observe changes that are being made so that they

can take these into account. Clearly, if the document was locked – such as with serialisable transactions – these engineers would have to wait until the document lock was released. This is unacceptable since it may unduly delay the overall development process. For this reason, mechanisms to allow *gradual relaxation* of the isolation property are needed. This means that requirement TR2 must be satisfied.

Further, the fact that engineers need to "protect" their code until some stage of the development process calls for rules and mechanisms allowing them to define and determine suitable *access rights.* This is also necessary when a document is released to restrict the actual access to a specific group, e.g., direct members of the module production. This means that requirement TR7 must be satisfied.

Case 2: Module test

Suppose the project has reached the test phase. Each module is tested by a team of several engineers. Assume that no bugs were found in the interface module, but that parts of its base class do not fully satisfy its requirement specification, and must be modified. Moreover, the two other modules use this interface base class, and are thus affected. Some minor modifications are therefore needed. However, the only way to allow the other teams to discover this problem is by making them aware of this explicitly, so a notification mechanism is needed. This means that requirement TR5 must be met.

They also need to know exactly which parts of the modules must be changed and how to do this. To help them with such tasks, it is useful to know what actions they previously have performed on the modules. This requires an *activity history* – i.e., an "activity log", thus calling for *temporal data management* satisfying requirement TR6.

Now, as the GUI and Process teams are aware of the changes being made, they have two options: either wait until the changes are made permanent, or observe the progress, make the necessary modifications, recompile, and test in parallel with other "processes". If we assume that waiting is unacceptable, it is necessary to have mechanisms that both allow the teams to be aware of the changes and allow them to perform their updates in accordance with the new base class. Note that both teams need to work cooperatively on the interface module, and thus none of them can lock it while they are performing their modifications. This exemplifies situations where one needs transaction models going beyond standard isolation – i.e., meeting TR2.

Case 3: Code modification

Cooperative code modification often lasts for long periods of time (hours or days). Much work could be invested during such periods. Therefore, if an engineer needs to cancel some actions, it would be unacceptable to also throw away other related parts that could or must be "saved". Moreover, it would be even more unacceptable that other engineers' tasks be affected by this cancellation. Therefore, the ability to provide a *finer abort management scheme* is necessary. This means that TR1 must be fulfilled.

Moreover, if an engineer is not able to accomplish a coding process, it would be useful if parts of the work or the results achieved so far could be handed over to other engineers in a controlled but flexible manner. This calls for mechanisms and tools to make such *delegation of responsibilities* possible – i.e., meeting TR3.

Case 4: Code inspection

Aviasoft's code inspection policy is to ensure as objective results as possible. For this reason, the inspection team consists of engineers from other divisions than the programmer's division. The team's tasks include not only checking the code for bugs, but also making sure that the code conforms to the underlying requirement specification, design and programming "procedures" for aviation software. For this reason the team has to access several types of documents including the code documentation from the Aviasoft's Intranet Web servers, the requirement specification documents from its file system, as well as the program code. To save money and exploit Aviasoft's networking facilities, the members of the inspection team are spread over the network, performing their tasks in a distributed fashion. Hence, to make this possible the underlying environment support has to make such access to *distributed and heterogeneous document*s possible – i.e., fulfilling TR4.

Case 5: Agenda update

Modifications to Aviasoft's project agenda are normally of short duration. This is done by computers and does not involve user interactions. Since an agenda normally contains the list of actions each engineer must perform, such updates must be done in a way that gives correct results. Users are often required to temporarily suspend their activities to alleviate them from doing unnecessary work. For this reason, both *atomicity* and *isolation* may be required. Here, atomicity ensures that either an update is executed successfully, or the agenda is left unchanged. Isolation means that each engineer should only observe a final version of the agenda. Hence, all other involved activities must temporarily stop until the update is accomplished. However, to let all involved parties know about such an agenda update, they must be notified. This case il-

lustrates a situation where full ACID transactions are also required in advanced applications like software development, in addition to traditional business applications such as accounting and banking. It also shows that even simple awareness services could prove useful. Ergo, again TR5 must be met.

5.5. Additional requirements

There are additional requirements that have been considered but not integrated in this work because of their scope. An important requirement is the support for version control. Let us, for example, consider the following case related to our Aviasoft scenario. A version x of a CFIS module is in the code inspection phase, while a successor version is in a requirement specification phase and a new code is generated from version x. This means that several versions of the same module may be in use so that the underlying consistency handling tools must consider their co-existence/co-activeness. For this reason, mechanisms for dealing with issues such as concurrency and atomicity should be provided that take other versions into account.

Although the importance of version control is recognised by this work, there are challenges that must be dealt with. First, the notion of version mutability may pose some difficulties. Several versions of the same object can be created by several users. This would challenge consistency management as it is not always possible to identify which versions are consistent and which are not. A possible solution is, of course, providing merging mechanisms, which could be used to merge some specific versions into a single final one. A challenge that this again may cause is the provision of a merging mechanism that is powerful enough, but at the same time does not result in computation complexity. Nevertheless, several approaches applying versioning and version control exist – including the NT/PV transaction model (Korth and Speegle 1994) and EPOS (Conradi et al. 1995, Conradi et al. 1997) – and this work could benefit from their adoption.

5.6. Concluding remarks

The key lesson we learned from setting up the above requirements is that it is hard to develop a single transaction model that can adequately meet the requirements of cooperative work. A possible solution is to provide a framework that allows customisation of transaction models to fit different situations and applications. This has been the main motivation for the development of our CAGISTrans framework, which is thus a framework for adaptable transactional support for cooperative work. For simplicity, we hereafter call this a transactional framework. A transactional framework is a vehicle used to de-

fine, analyse and apply the essential properties of transactions. The next chapters will elaborate on our CAGISTrans framework.

Chapter 6
The CAGISTrans Transactional Framework

6.1. Introduction

This chapter gives an overview of the ideas behind the CAGISTrans transactional framework. It focuses on ways to support cooperation that is dynamic and heterogeneous, using transactions. First, Section 6.2 outlines a proposed solution to support object and data exchange through workspace management. Second, using this as a starting point, Section 6.3 proposes a new way to organise the transaction modelling process to meet the dynamic requirements of cooperative work. It discusses how this can be achieved by separating design time and runtime specification of application specific transaction models. Third, as the main result of the distinction discussed in Section 6.3, Section 6.4 outlines how transactional behaviour can be managed at runtime to further meet the requirements imposed by dynamic environments. This includes a discussion of how user-managed transaction correctness criteria can be achieved, as an alternative to serialisability (Bernstein et al. 1987). Fourth, to meet the aforementioned heterogeneous requirements, Section 6.5 discusses a middleware approach to provide the required transactional service. Fifth, Section 6.6 discusses the need for a specification language that can enable runtime specifications. We focus this discussion on XML, including the main advantages and the challenges that must be taken into account.

6.2. Bridging the gap between flexibility and strictness using workspaces

A widely accepted solution to support cooperation is to provide workspaces – i.e., virtual spaces or places where groups of distributed people can cooperate in solving a problem or performing a task. Usually, one distinguishes between private and shared workspaces. This makes it possible to alternately perform work in cooperation, and carry out individual activities in private. Such a separation has been considered important since it is neither desirable nor practical to share all data all the time. In effect, control and management of workspace access is necessary.

However, in existing systems, the main problem is that either the control and management are left to the users themselves to figure out – cf., groupware-based approaches, or they are unduly strict – cf., database-centred approaches. To bridge this gap, there is a need to organise the workspaces in such a way that different levels of sharing can be enabled. This can be achieved by providing workspaces with *nested structure*. Moreover, operations to allow access to workspaces at different nesting levels are needed. This is achieved by extending the traditional check-in/check-out models (Tichy 1985, Rochkind 1975) with *advanced check-in and check-out operations*. Finally, mechanisms to synchronise and coordinate the involved accesses must be provided.

These are the workspace features that will be discussed in the following. Their relation to other work is discussed in Chapter 11. In summary, their main benefit is the ability to control the access to workspaces, while at the same time increasing the possibility of sharing, compared to traditional check-in/check-out models.

6.2.1. Workspace organisation

Aviasoft's (see Section 5.1) software development process involves several engineers. Because of the size and complexity of the CFIS software artefact to be developed, the engineers are organised into small groups developing different but related modules. To allow flexible cooperation, engineers belonging to the same group have to be supported by a shared workspace where they can exchange their document – i.e., program codes, etc., specific to their module. However, until reaching some stage of their coding process, each engineer needs to work in private. This is true, for example, when the code that is being written is still so incomplete that sharing would not be reasonable. To cope with this, each engineer must have a private workspace. Further, when a module reaches some maturation phase, it may be of interest for other related groups to have access to what has been achieved so far. Therefore, the module

and all its related data can be made available by copying them to a public workspace. This could, in addition, be necessary when the modules are to be integrated into a single software artefact.

As can be inferred from this simple illustration, at least two levels of data sharing may be needed: sharing among engineers within the same group, and sharing among and across different groups. As a result, distinguishing between private and public workspaces alone would not be sufficient. In addition, *group workspaces* may be needed. This means that workspaces must be organised, as illustrated in Figure 6.1, to form a nested structure, with unlimited nesting levels. The main idea is to compromise on isolation among workspaces, but at the same time limit or avoid unintended and unauthorised manipulation of data at a specific workspace.

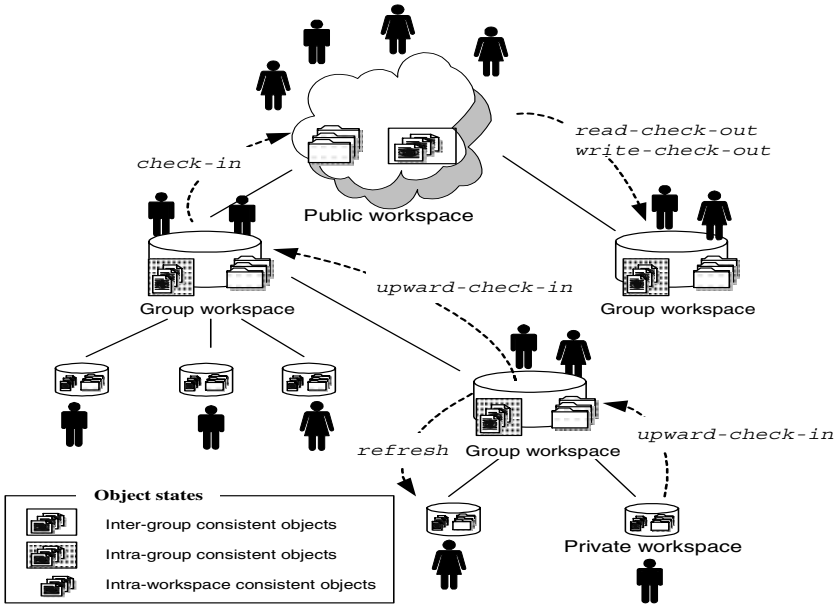


Figure 6.1 Nested workspace structure with corresponding operations and object states.

Another aspect is the management of objects. As a supplement to versions, we may apply different levels of object consistency through object states. The idea is that each object is associated with a state depending on the nesting level of the workspace where it resides. This means that we distinguish between *intra-workspace consistency*, *group consistency* and *inter-group consistency*. In other words, objects in private workspaces are only *intra-workspace consistent*. When they are released or copied to a parent – i.e., group – workspace, their consistency level is upgraded to *group consistency*. Since such objects may still be inconsistent with respect to the final results – e.g., the final software artefact, sharing is restricted to members of the group who own that workspace,

only. Finally, when the objects are checked in to the public workspace, they are considered *inter-group consistent*.

6.2.2. Extended workspace access operations

An extension of the traditional check-in/check-out model is required as the main consequence of the workspace and data organisation. This extension serves as a means to manage the relevant object states. Moreover, additional operations can be used to distinguish between submitting objects to a sub-workspace and releasing objects to a public workspace.

The following operations are needed in addition to check-in and check-out:

- *write-check-out* and *read-check-out*: distinguish the intention behind the check-out operation. They check-out data from the public workspace for write and read, respectively. The main advantage is that this facilitates the control of sharing, and allows the use of appropriate lock modes.

- *upward-check-in*: check in data to a parent group workspace – i.e., up one level only.

- *check-in*: check in data from any workspace to the public workspace.

- *refresh*: update a copy of data in the private workspace with the one residing in the parent workspace – e.g, check-out from a group workspace to a private workspace.

- Workspace data operations such as read, write, update (i.e., read and then write), and some advanced operations (see Section 6.4) which are needed to manipulate data at a specific workspace. Advanced operations include those needed for consistency level upgrade, after an upward-check-in.

Note that the ideas of extending the standard check-in/check-out model share some ideas with those previously proposed in the literature (Kim et al. 1984). For example, the idea here of allowing access to data in semi-public workspaces (group workspaces) is similar to that approach. However, Kim et al. (1984) consider the direct connection between workspace access operations and the nested transaction structure. For example, this approach distinguishes between *downward commit* and *upward commit*. *Downward commit* allows transactions to check-in data to a semi-public database such that they can be check-

out by other transactions. Transactions that check out these data become, by default, subtransactions of that issuing the *downward commit*. These subtransactions can then use *upward commit* to check in data to the semi-public database of their ancestors. In this sense, *upward commit* and *downward commit* only apply to transactions that are family related. By contrast, the *upward-check-in* operation used in the CAGISTrans framework can be issued by arbitrary transactions that are not necessarily family related. The only requirement here is that these transactions are executed by people sharing a group workspace.

6.2.3. Workspace access coordination

Correctness of each object state can be managed by synchronising and coordinating the access to workspaces. To facilitate this coordination, workspace operations can be executed as part of a transaction. Hence, consistency of objects that are shared among several users can be controlled by indicating necessary operations to be carried out, and enforcing the order in which they are performed. This implies that if transaction executions obey such criteria, it may be assumed that their results are correct. How this is done is one of the main topics of the rest of the thesis (see Section 6.4).

6.3. Distinguishing between characteristics and execution specifications

One of the main limitations of the transactional frameworks that were developed for cooperative work is their restricted support for changing environments – i.e., dynamic support. Despite their ability to define transaction models for specific applications, they generally lack the ability to make adjustment and modification at runtime. One of the main reasons is that a complete specification of a particular transaction model must be done before a transaction based on that model can be executed. This is, however, practical only for applications/situations where all necessary requirements are possible to predict in advance. Normally, in cooperative work, this is not the case. For instance, in modern software engineering environments, the requirements may change all the time. So, to meet these changes, the model needs to be modifiable at runtime.

6.3.1. The necessity of design time and runtime specifications

Modifiability can be achieved only by allowing both *design time* and *runtime* specifications. This means that we need to organise the specification of the transaction model into two separate but connected parts, consisting of one that can be designed before any transactions are executed, and another that can be modified while the involved transactions are being executed. The former is

called characteristics specification and the latter is called execution specification. This distinction is illustrated in Figure 6.2.

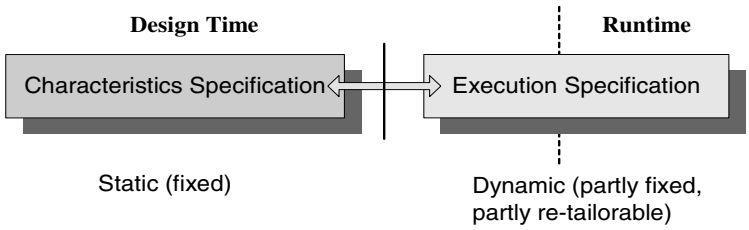


Figure 6.2 Illustration of the distinction between characteristics specification and execution specification.

The idea is to collect elements of a transaction model that are fixed and possible to predict in advance into the characteristics part, and include the elements of the transaction model that are only partly predictable into the execution specification. In this respect, the transaction characteristics consist of the elements that are vital and must be known prior to the execution of transactions. First, they specify, at a high level, the main properties of the involved transactions, i.e., the ACID properties. Second, they determine suitable structures of the transactions, and define how they affect each other's processing, i.e., the relationship among the involved transactions. Third, the elements define appropriate correctness criteria to be applied. This means that they specify whether one should rely on an underlying DBMS and/or enable user-defined criteria. Finally, the characteristics elements determine the mechanisms and policies to be utilised in order to satisfy the designated correctness criteria. In other words, they define mechanisms that should be made available, and the rules and guidelines for how and when to use them to achieve the required correctness.

As mentioned above, the transaction execution part is a collection of elements that are most often only partly possible to specify in advance. For this reason, they may need to be adjusted at runtime. Further, they determine how transactions having the specified characteristics should behave during runtime. This means that the execution specification must be composed of building blocks which, together, satisfy the requirements imposed by the characteristics specification. In addition, the execution specification extends the characteristics specification since it contains elements that are not included in the characteristics specification. Thus, the idea is to collect all elements that affect the trans- actional behaviour into the part that is specific for transaction execution. Briefly, these are primitives that are needed to manage and control the execution of transactions - i.e., initiation and termination operations, those needed for dynamic restructuring of transactions at runtime - i.e., spawning

and resource transfer, and those used to define advanced operations that will be executed by the transaction. A more detailed discussion is given in Section 6.4.

6.3.2. ACID requirements and their impacts

This discussion will be centred on the elements of the characteristics specification. It analyses the transactional properties in more detail, and shows how the rest of the elements of the specification are derived.

The first issue concerns the reasons for explicit specification of ACID properties. Since (Härder and Reuter 1983) coined the ACID acronym, it has been used extensively within the database community as the ultimate concept for achieving correctness of transaction processing. As mentioned before, however, when used in advanced applications, such as cooperative applications, some of the properties have to be compromised in order to provide the required support. Which of the properties that should be relaxed depends very much on the nature of particular applications. Therefore, it is necessary to explicitly specify each of the properties in order to tailor them for specific applications or situations.

The main advantage of such an approach is that allows the user to identify the main characteristics of the application he/she is involved in. For example, some situations within advanced applications may still require atomicity to guarantee the correctness of data in the event of crashes. Others may see it as a burden since it might require people to unnecessarily repeat work that has already been considered finished. Similar arguments can be used for the remaining properties. So, the idea is that users do the designation themselves by first identifying application needs, and then defining appropriate "ACID properties".

This leads us to the next issue: what properties should be compromisable and how can that be implemented? From the literature, the "porting" of ACIDity from traditional database applications to advanced applications has implied compromising the atomicity and isolation properties only. It seems that it is widely accepted that both consistency and durability should still be preserved (Elmagarmid et al. 1992, Kaiser 1994, Jajodia and Kerschberg 1997).

6.3.2.1. Atomicity

Starting with atomicity, there should be two options: full atomicity or relaxed atomicity. Going for full atomicity should be possible, since it may still be vital to preserve the all-or-nothing property of transactions. This may, for instance, be relevant in activities that do not involve user interaction, and are

short. The idea is that atomicity should be kept whenever it is appropriate. Relaxation of atomicity, on the other hand, may be required for several reasons. First, the presence of failures in long running activities may be expensive since lots of work could be thrown away. Therefore, it is necessary to compromise on the all-or-nothing law. Thus, relaxed atomicity should serve as a means to allow finer grained abort management in the form of partial abort. Further, it is common in cooperative environments that users choose the actions within their activities as they go along - i.e., interactive transactions. In such a case, requiring transactions to perform automatic abort is inconvenient. This means that users should be able to control the rollback themselves. However, this may be achieved only if the atomicity property is relaxed.

To achieve relaxed atomicity by means of partial rollback, a user or a transaction model designer needs to specify the desired structure of the transactions to be run. It means for example choosing explicitly between flat and nested, and the intended dependencies among transactions. First, relaxed atomicity is most relevant for transactions with a nested structure. Therefore, to allow partial abortion, the choice will fall on nested structure. Second, the effect of the abortion of one transaction on others needs to be specified. This means that a user or a transaction model designer needs to define the *abort dependencies* (Chrysanthis and Ramamritham 1990) among the involved transactions. This will explicitly determine the transactions that have to abort, regardless of their read-write dependencies. Hence, partial rollback is accomplished by first checking the abort dependency specifications, and then aborting only (sub-)transactions that are included in those specifications.

Table 6.1 provides a summary of the relevant elements impacted by the value of the atomicity property.

	Full Atomicity	**Relaxed Atomicity**
Transactional relationship	Flat or Nested structure Inter-transaction dependencies: begin, commit and abort	Nested structure Inter- and intra-transaction dependencies: begin, commit and **abort**[a]
Correctness criteria	Depending on isolation, see Table 6.3	
Applied policies	Depending on isolation, see Table 6.3	

Table 6.1 Atomicity and relevant elements.

a. Boldface means the dependencies are mandatory.

It is worth noting that some of the above ideas concerning nested transactions and abort management can be compared with some of the basic ideas of

HiPac (Dayal et al. 1988). In general, ideas related to event-condition-action (ECA) rules can also be applied here to specify (1) the abort-events, (2) the condition for these events in terms of abort dependencies, and (3) the actions to be taken – i.e., aborting affected subtransactions – determined by the abort dependency specifications. Further, the idea of failure management in the CAGISTrans framework has some similarity with that of HiPac (Dayal et al. 1995) in that partial rollbacks are managed through exploiting the nesting structure of nested transactions. This means that when a specific subtransaction fails, the effect on other transactions can be determined depending on existing conditions.

Despite the above similarities, there are ideas of the CAGISTrans framework that differ from those of HiPac. First, although the active rules of HiPac can also be used in the specification of transaction models – such as with TSME (Georgakopoulos et al. 1996) – supporting transaction model customisation itself was not the primary focus of HiPac. Moreover, its specific way of managing rollbacks is different from that of CAGISTrans. For example, with HiPac's nested transactions, when a subtransaction fails, this failure is by default propagated to one or more ancestors, which then decide appropriate further actions. With CAGISTrans a more general abortion scheme can be achieved (see also Section 7.2.3). Another difference is that in HiPac a failure of a subtransaction only affects other sibling subtransactions that are specified in their parents' event rules. This means that the parent transaction is fully responsible for handling the abortion of subtransactions. In CAGISTrans, a parent transaction does not necessarily have such responsibility. Moreover, abortion of a subtransaction can be specified to affect other specific siblings, independently of their parents.

Regarding the distinction between characteristics specification and execution specification, note that atomicity mainly affects the way transactions achieve their specified structure. Recall that transactions with relaxed atomicity are nested. This means that it is likely that some of these transactions need to restructure – i.e., spawn and delegate some responsibilities – at runtime. Therefore, to allow this, a primitive for transaction restructuring should form part of the transaction execution building blocks.

6.3.2.2. Consistency

Consistency needs to be fully preserved to ensure the correctness of the final result of a transaction execution. Though some kinds of inconsistency may have to be tolerated by transactions executed in cooperative environments, who is interested in introducing inconsistencies in a database or other involved resource base? Further, this is one of the main reasons that research on transac-

tions is relevant at all, also within the context of cooperative work. The concept of transactions is meaningless if consistency cannot be achieved in the final results.

However, there is a challenge that needs to be considered. The relaxation of the atomicity property implies that a transaction may commit parts of its total processing. A question that may arise is then how to define the correctness of such a transaction? As an answer to this question, specific the correctness criteria should be prerequisites. Usually, a minimal set of criteria must be defined before a transaction is executed. This means that either a user defines an appropriate criterion or that an underlying DBMS provides one – e.g., serialisability. Table 6.2 summarises the relevant elements concerned with ensuring consistency. Full consistency implies that if an underlying DBMS does not provide any suitable correctness criterion, the user must specify the constraints to be enforced on the execution of transactions. This means that the execution specification must contain a block that allows the definition of such constrains.

	Full Consistency
Transactional relationship	Depending on atomicity, see Table 6.1
Correctness criteria	Either database dependent or user-defined, see Table 6.3
Applied policies	Depending on applied correctness criteria

Table 6.2 Consistency and relevant elements.

6.3.2.3. Isolation

As for the atomicity properties, we should provide two isolation options: full isolation or relaxed isolation. Full isolation is still required in some situations where sharing is not a prerequisite. For example, in the cooperative programming environment in Section 6.2, one would need to work in private until the source code has reached some maturation phase such as the first release. However, often in the context of cooperative work, full isolation would contradict the required level of sharing. Moreover, transactions executed in the context of cooperative work normally have long duration. Therefore, requiring other transactions to wait until specific transactions commit would be unacceptable.

Thus, to achieve relaxed isolation there is a need to specify and apply correctness criteria that are more relaxed than serialisability. As a result of the above mentioned consistency requirement, appropriate constraints must be added. Further, to allow sharing among transactions, a sharing primitive must be provided to explicitly specify the interleaving that is allowed. To achieve the criterion specified, policies defining mechanisms and the corresponding rules

for their usage should be provided. For instance, diverse locking types (isolation) can be combined with awareness primitives – i.e., notification. In order to cope with the problems that possible cascading aborts may cause – like aborting all transactions that share the same data, relaxed isolation (further) motivates the need to explicitly define the *abort dependencies* that apply on the involved transactions. In other words, if a transaction fails, not all cooperating transactions have to be affected. Only those covered by such abort dependencies are required to rollback.

Table 6.3 provides a summary of the relevant elements impacted by the value of the isolation property.

	Full Isolation	**Relaxed Isolation**
Transactional relationship	Depending on atomicity, see Table 6.1	Depending on atomicity, see Table 6.1
		Mandatory inter- and intra-transaction abort dependencies
Correctness criteria	Database dependent	User-defined
Applied policies	Strict locking policy	Collaborative locking policy and awareness

Table 6.3 Isolation and relevant elements.

As shown in Table 6.3, abort dependencies are necessary when the isolation is relaxed. The main reason for the connection between abort and isolation is to allow flexible handling of aborts that may result from the sharing of data. Abort dependencies are only applicable where one or more transactions simultaneously access shared objects. However, at a specific time, transactions may be isolated (no sharing). Later, they may become cooperative, thus sharing some specific objects. (Such situations may, for example, occur in version based systems). Here, the dependencies will first be applicable from the time these data are concurrently accessed by two or more transactions. Hence, assuming that all objects that are made accessible to other transactions are either committed (checked in) or pre-committed (upward-checked in), the abort dependencies here will not be affected by the occurrence of isolation of execution at a point in the past.

With respect to the distinction suggested in this thesis, there is an implication similar to that of the atomicity property. As already mentioned, relaxed isolation requires that a user needs to specify correctness criteria. This means that specification of how transactions view and affect other's processing must be given. More details about how this is achieved are given in Section 6.4.

6.3.2.4. Durability

As for the consistency requirements, the final results of transactions must be permanent. Provision of persistence for committed data is one of the key philosophies underlying DBMSs. The main idea is that once the results are committed, it should survive crashes, and other similar system failures.

One important question that may arise is, when to save data in long running activities that may involve cooperation? It is reasonable to assume that long-running transactions are split into several sub-transactions, resulting in a nested structure. Consider that each sub-transaction wants to commit its results. These results can be made permanent by saving them in the underlying repository. But, how such commits affect the final results depends on the *commit dependency* specifications for the involved transactions. Further, as a result of relaxed isolation, sharing may occur. To handle abort propagation in such cases, *compensating transactions* – cf., (Korth et al. 1990, Levy et al. 1991) – must be provided to explicitly discard the effect of committed results. Hence, *abort dependencies* among transactions must also be defined and to be able to fulfil the durability requirement, all transaction operations must be logged. This means that either an underlying DBMSs does the logging, or a supported mechanism describing the transaction execution is provided. Table 6.4 summarises the relevant elements concerned with ensuring durability.

	Full Durability
Transactional relationship	Flat or Nested structure Inter- and intra-transaction dependencies: begin, **commit**, and **abort**[a]
Correctness criteria	Depending on isolation, see Table 6.3
Applied policies	Depending on correctness criteria

Table 6.4 Durability and relevant elements.

a. Boldface means the dependencies are mandatory.

6.4. Dynamic re-specification of transactional behaviour

A widely accepted solution to overcome the diverse nature of cooperative applications is to have transaction models that can be adapted to different needs. The dynamic properties of the involved cooperation environments have, on the other hand, broadened the scope of adaptability to comprise modifiability and extensibility. New requirements may occur while transactions are being executed. Therefore, the specification of a transaction model must be adjustable at runtime. In particular, this affects the definition of how transactions

behave. Normally, it is not practical or possible to plan complete transaction schedules a priori due to the aforementioned dynamic properties. This also means that transactions are normally interactive. This makes it important as well as challenging to allow such redefinition.

The ability to modify and adjust the specification of transactional behaviour at runtime yields several advantages. First, this may overcome the restriction caused by dependencies on a complete knowledge of actions. In other words, as extensions and adjustments can be accomplished while the involved transactions are being scheduled, users do not need to worry about setting up complete actions to be carried out in advance. Moreover, as users can do part of the specification themselves – as their activities go along – they have better control of the execution of their (trans)actions. This means that a user can, to some degree, choose the necessary actions to attain his/her goals, including transaction termination and restructuring. This, in turn, implies that open-ended and long running activities may be supported.

To achieve all this, we need make the distinctions shown in Figure 6.2. In addition, there is a need to separate the execution specification into two main parts: fixed execution specification and modifiable execution specification, as depicted in Figure 6.3. In this way, predictable execution primitives can be collected in a fixed part, whereas those that may have to be modified at runtime are included in a modifiable part.

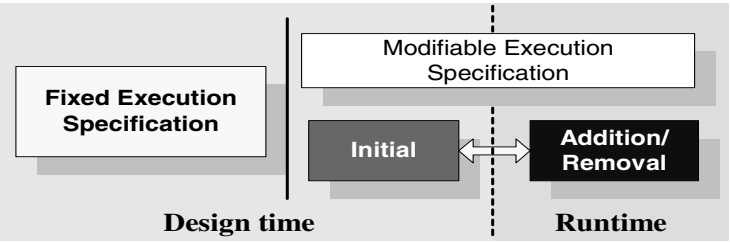


Figure 6.3 Illustration of the distinctions in the transaction execution specification.

6.4.1. Managing transactional behaviour

The building blocks of the execution specification that affect transactional behaviour are as follows (see also Figure 6.4):

1. *Transaction constraints*, which are used to control and manage the effects of the execution of transactions on other transactions.
2. *Management operations*, which are used to control and manage the initiation, termination, and dynamic restructuring of transactions.

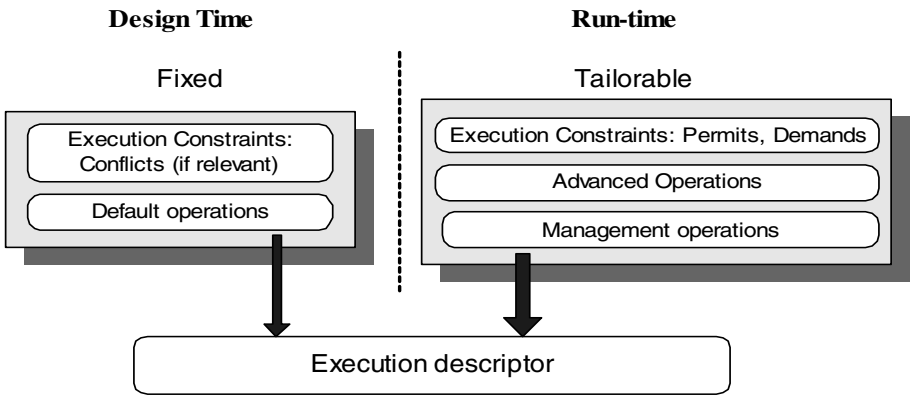


Figure 6.4 Elements of the execution specification.

3. A*dvanced operations*, which are used to specify operations at higher abstraction levels than read and write.

4. *Transaction execution descriptor*, which contains all the information on the execution of transactions as well as information related to transaction model specification.

This discussion is centred on these primitives.

6.4.1.1. Effect management using transaction constraints

The effect of a transaction on other transactions depends on the interaction among the involved transactions. It can be managed by defining constraints that explicitly determine the interleavings that are a) *forbidden*, b) *allowed*, and c) *mandatory*. Each of these constraints are elaborated below. Table 6.5 summarises this discussion.

	Purpose	**Relevant when**	**Not Needed when**
Conflicts	To hinder interference	Possible to predict a set of actions	Impossible to predict a complete set of actions
Permits	To allow flexible sharing	Isolation is relaxed, and conflicts are defined or locking is used	Isolation is full, or no conflicts are defined and no locking is used
Demands	To ensure correct execution	Always relevant when isolation is relaxed	

Table 6.5 Summary of effect management.

Conflicts

Constraints identifying operations that are not allowed to execute concurrently are referred to as *conflicts* – as in (Nodine and Zdonik 1992). Their main

purpose is to hinder interference by forbidding specific interleavings to occur. Hence, an operation is allowed to execute only if it obeys the conflict rules. In this sense, conflicts can be seen as an analogy to locks.

Now, the question that may arise is: when are conflicts relevant? Conflicts should be made available if flexible locking mechanisms – i.e., those allowing different degrees of sharing – are not supported. Clearly, to be able to hinder interference, a system must provide protocols or mechanisms to manage concurrency. At the same time, it is important that such mechanisms and protocols are flexible enough to allow cooperation. Therefore, as long as it is possible to predict the set of actions to be carried out within the cooperative activities, the definition of conflicts is encouraged. Because conflicts may be customised to the application needs, they may generally allow a higher degree of flexibility than standard locking protocols – e.g., 2PL (Bernstein et al. 1987).

But, should conflicts be fixed or modifiable? The answer lies in finding a trade-off between manageable flexibility and "anarchy". Manageable flexibility can be achieved only if conflicts are fixed. This means that if conflicts are relevant, they must be specified before the involved transactions are executed. However, to cope with the limitations caused by the dependency on complete a priori knowledge of actions, conflicts can be supplemented with a locking protocol that allows users to issue locks on their behalf as their activities are in progress. Such type of locks are called *user-controlled locks*, and are similar to those utilised in existing cooperative systems – e.g., BSCW[1]. Our user-controlled locks differ from existing ones in that instead of totally relying on users to issue all locks, they serve as a supplement to conflicts. Hence, freedom may be restricted to a manageable level.

The idea is that user transactions do not have to follow strict protocols that require automatic acquisition of specific locks for specific operations. Instead, at some point in the cooperation process, a user can lock the objects he/she wants to control access to. This means that if the rules defined by conflicts are incomplete, user-controlled locks may be applied. On the other hand, if conflicts are too restrictive with respect to cooperation, "allowed interleavings" is relevant.

Permits

Such constraints define relationships among operations that are normally conflicting – such as according to the conflict rules – but are allowed to appear, anyhow. These are referred to as *permits* – see (Biliris et al. 1994). Their main purpose is to allow controlled but flexible sharing. Broadly speaking, permits

1. See http://bscw.gmd.de.

define a set of relationship identifying transactions that are allowed to cooperate on a set of data, whether their operations are conflicting or not.

As in conflicts, questions of relevance and modifiability may arise. Initial permits can be defined before runtime to allow a set of transactions to override a possible conflict definition. This means that, based on permits, some executing transactions performing conflicting operations are granted access to the same data, but others are prohibited. Further, due to the interactive form of transactions, it is not always possible to predict a priori the interaction or interleavings among transactions that are needed at runtime. Therefore, the ability to modify permits at runtime is required. This includes the *addition* and *removal* of permits. New permits may, for instance, be needed when new transactions are initiated. This is normally unproblematic, since it will only affect future execution of actions. Conversely, a situation in the cooperative process may cause permission to share being irrelevant or undesirable. Therefore, the ability to temporarily or permanently remove some specified permits must also be considered. The removal of permits may imply that interleavings that have already appeared become illegal with respect to applicable conflict rules. This, in turn, may result in invalidation of already performed actions, thus wasting a lot of work. Therefore, it is necessary to make sure that removal of specific permits does not cause expensive invalidation of actions.

Although the introduction of permits does not affect already executed actions, this may allow concurrency anomalies such as dirty reads and lost updates, that must be taken into account. Dirty reads are the least serious of these, referring to reads of uncommitted data. Although they are generally undesirable, avoiding dirty reads would contradict the whole philosophy of permits. Nevertheless, the usage of permits has to be managed explicitly in order to avoid inconsistency of the final results. As a solution, the definition of permits is accompanied with specification of *dependencies* among the involved transactions – cf., relaxed isolation in Section 6.3. In addition, there is a need to apply *notification mechanisms* that make users aware of the effects of particular sharing situations, and help them manage and adopt vital changes into their transactions.

The other anomalies should normally be avoided. Lost updates occur when a transaction writes data that are later directly updated by another transaction. Such situations may occur when permission to overwrite data is given. A simple workaround is to prohibit overwrite situations. But this may cause blocking. A solution is to allow collaborating transactions to hold locks that are normally considered conflicting – i.e., similar to those applied in Solaris[1] file

1. Solaris is a trade-mark of Sun Microsystem.

systems, but at the same time control simultaneous updates on data. Such locks are referred to as *collaborative locks*. As with user-controlled locks, such lock types are managed by users. In contrast, while user-controlled locks avoid all types of conflicting situations, collaborative locks only control overwriting situations. In other words, when locks are acquired, other users may acquire other conflicting locks. But, in the case of concurrent updates, the involved users will be warned, requiring them to choose the way to resolve the specific conflicts on their behalf, before any further action is taken. Other restrictions that apply are determined by the existing *commit dependencies*, – such as determining the valid commit order.

Demands

Constraint c) defines the interleavings that are required for transactions to produce a correct execution. These are referred to as *demands*[1], defining sequences of operations or steps that must appear. Considering the required dynamic behaviour of transactions, specification of demands should be modifiable. Normally, only part of such a specification can be done before runtime for the same reasons as permits. In this respect, modification of demands includes removals and additions of constraints. Removal of demands means that required actions are no longer needed. This is normally unproblematic since it will not affect already performed or committed actions.

Additions mean that new actions are inserted into the demands specifications. This may be needed when new transactions are initiated, or when new steps are required. In contrast to removal, inserting new constraints may impose some difficulties. For instance, new actions may appear between two already executed steps. Therefore, the addition could be useless. Alternatively, it might imply that results from executed transactions are no longer correct with respect to the new demands. The addition of new constraints is restricted to involve actions that have not been executed yet – checked using the *execution descriptor*. If new steps appear between already executed actions, such actions may have to be aborted, and re-executed to reflect the new constraints.

6.4.1.2. Management of transaction execution

Management operations consist of primitives that are used to manage and control the initiation, termination and dynamic restructuring of transactions (see Table 6.6). Initiation of transactions is done by issuing *begin*. To terminate initiated transactions, either *commit* or *abort* is issued. Due to the required user control, transaction terminations are usually interactive. But, the

1. Demands are built on *patterns* (Skarra 1989, Nodine and Zdonik 1992) with modifications and extensions.

	Operation types	Usage and effects
Control of execution	Initiation: `begin`	User-controlled if relaxed atomicity
	Termination: `commit` or `abort`	User-controlled if relaxed atomicity. The effects on other transactions depend on commit and abort dependencies
Dynamic restructuring	Delegate for flat transactions	Transfer of operations only
	Delegate for nested transactions	If target child(ren) exists, then operation transfer only
		If target child(ren) does not exist, then spawning before operation transfer

Table 6.6 Summary of execution management operations.

effects of a termination on other transactions depend on specified dependencies – i.e., commit and abort dependencies. This means that if a user terminates a transaction he/she will also be informed which other associated transactions may have to be terminated. Note that due to the incurred "cascading aborts", a list of affected actions is provided informing the user about this effect, so that he/she may first choose to commit partial (possibly finalisable) results, thus minimising the cost of the aborts. Users may not always be able to understand the real cost of such aborts – e.g. in terms of wasted person-hour work. Therefore, finding an efficient way to provide users with as accurate information as possible concerning the above cost is an important challenge that must be still considered.

The concept of the dynamic restructuring of transactions has been recognised as an important primitive to allow cooperation among transactions (Kaiser 1994). The concept can be realised in several different ways, with possibly different purposes (see below). The use of dynamic restructuring of transactions is stressed here to realise a specified transaction structure (e.g spawning new transactions) and allow transactions to delegate responsibilities to other transactions at runtime. To this extent, the relevance of dynamic restructuring depends on the defined structure for the transactions. In other words, full restructuring can only be achieved if the involved transactions have a nested structure, as defined by the characteristics specification. This means that if a transaction is flat then only operation transfer is allowed, since spawning is irrelevant. If the transaction, on the other hand, is nested then dynamic restructuring may be accomplished in two ways. Operations can be transferred to existing sub-transactions. Or, if a target does not exist, creation of a new subtransaction precedes the intended operation transfer.

6.4.1.3. Support for advanced operations

The idea of advanced operations is to allow the specification of operations at an abstraction level higher than, but based on, *read* and *write*. Such an approach is useful because of its ability to exploit the operation semantics to increase concurrency (Korth 1983, Weikum and Schek 1992). Moreover, abstraction beyond read and write is required for advanced applications, such as software engineering environments and the like (Ramamritham and Chrysanthis 1997). Modelling software engineering activities – e.g., compile, edit, and so on – with read and write is generally a complex task. Using advanced operations, we may increase the modularity and thus simplify the modelling task.

To this extent, one of the main goals of advanced operations is to further improve the usability of the transactions within advanced applications. However, there are issues that need further consideration. An important one is whether the definition of advanced operations at runtime should be allowed or not. As for transaction constraints, it is desirable to allow runtime definition. Recall that a complete set of tasks is not always possible to predict in advance. New operations may be required as activities proceed. The complexity of introducing new advanced operations at runtime may, however, impose some difficulties. First, introduction of new operations may require the definition of new *conflicts*. However, referring to Section 6.4.1.1, conflicts cannot be modified. Therefore, runtime definition of new operations is only allowed if it does not require new conflicts. Further, since no new conflicts exist, new *permits* are only necessary if user-controlled locks are used. Finally, the introduction of new *demands* may be required as new steps will be executed. The restrictions that this may imply are that new operations added to the actual demands should not appear between already executed steps (see also Section 6.4.1.1).

In conclusion, most of the advanced operations must be specified before the execution of transactions, but new operations may be introduced during runtime given that they do not need to modify conflicts.

6.4.2. Dynamic user re-definable correctness criteria

The discussions in Section 6.4.1.1 imply that user-defined correctness criteria constitute a combination of *conflicts*, *demands*, and *permits*. They serve as vehicles to ensure the correctness of transaction execution that, in terms of flexibility, go beyond serialisability.

First, conflicts are a set of application specific rules identifying forbidden interleavings to hinder bad interferences. Second, demands define the necessary sequence of steps that are required to appear, thus providing a possibility

to control the execution of transactions towards consistent final results. Third, permits serve as a means to weaken conflict rules and provide the possibility to allow flexible but controlled sharing among transactions. This is despite the fact that the inferred interleavings that may occur could have been considered illegal in traditional DBMSs.

The user-defined correctness criterion can be specified and tailored to a specific application. In addition, parts of this – i.e., permits and demands – can be adjusted and modified at runtime (see Figure 6.5). This means that an incremental specification is possible, thus limiting the requirements for predictability. This again implies that such criteria may fit well in continuously changing environments. Since the specification can be extended gradually, the problem implied by the accompanying complexity could be alleviated.

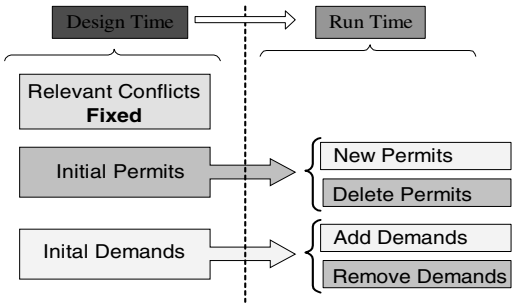


Figure 6.5 The mechanisms of the user-defined correctness criteria.

As can be inferred from the above discussion, the CAGISTrans framework has primarily focused on achieving correctness through execution constraints – rather than using data consistency predicates. The main reason for this is the emphasis on flexibility – though using such predicates also allow relaxed correctness criteria – which is one of the main requirements for cooperative work. Flexibility here means the ability for users to control the execution of transactions and tailor these to their needs. By providing transaction execution constraints, this can be achieved more easily than with data consistency constraints as a transaction model designer can specify and customise the constraints to be applied on the transaction execution according to the application needs. However, the present author also recognises the advantages of the latter approach. Allowing user-defined execution constraints may be subject to errors, which again could jeopardise the correctness of final results. Using data consistency predicates, on the other hand, we might cope with such error-prone specification tasks by allowing the system to do automatised control based on some specified data consistency predicates (Korth and Speegle 1994, Bancilhon et al. 1985), and relieving the users from doing the specifications

themselves. The cost that might have to be paid is, however, that developing a customisable framework would be a challenge as data constraints are defined implicitly and users cannot specify them to suit their specific applications. In conclusion, choosing between data consistency and execution consistency is a matter of finding an appropriate trade-off between (1) flexibility in term of adaptability and (2) error-free (automatised) correctness preservation.

6.5. Supporting heterogeneity

A main requirement for cooperative work is support for heterogeneity. The problems of addressing heterogeneity of resource management systems have long been a subject of intensive research in the database field. Examples of work in this respect are those on multidatabases focusing on transaction management over heterogeneous and autonomous DBMSs (Breitbart et al. 1995, Mehrotra et al. 1998, Mehrotra et al. 2001).

Despite the usefulness of the techniques from the multidatabase systems, only a few existing *customisable transaction models* have adopted their basic ideas. An example of those is TSME (Georgakopoulos et al. 1996). Morever, the support for other resource managers than DBMSs seems de-emphasised. This may be a shortcoming, since users in the cooperative environment may want to access information or data from several places that do not necessarily reside in a database. For example, data needed during a collaboration process may reside on a Web server, in regular file systems, heterogeneous database systems, or other storage services. For this reason, openness is crucial.

To cope with openness and other factors such as dynamic properties, a system supporting the CAGISTrans framework has been implemented as middleware bridging user-applications and resource bases (see Figure 6.6). The ad-

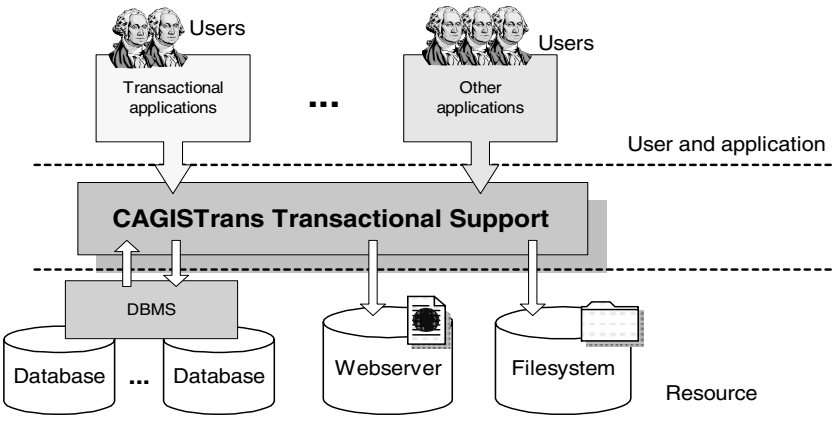


Figure 6.6 CAGISTrans high-level architecture.

vantage of this is that we do not have to build the system from scratch as in the case with TSME.

Because of this, CAGISTrans inherits several advantages from the well-known middleware approach, combined with those from the multidatabase system techniques:

- *Increased support for distribution* is achieved by means of Web support. The CAGISTrans middleware allows information and data that are administered by the transactional framework to be accessible from the Internet. Thus, people can work together independently of their geographical location.

- *Increased resource availability* is gained by allowing resources administrated by the CAGISTrans system to reside on repository types other than databases. This makes it possible to access richer information and data types than those supported by database systems. Such data may be of interest or even critical for the accomplishment of the cooperative process.

- *Database independence* is attained since CAGISTrans is developed as middleware rather than a complete transaction management system – e.g., an advanced TP-monitor. This is based on the observation that the transaction specification and execution can be accomplished independently of the underlying repository support. This also enables the porting of the transaction models to different types of systems, thus extending the usability of transaction models to wider application areas.

The main open question concerning this approach is the performance issue. As can be inferred from the discussion in Section 6.4, execution of such transactions may introduce some overheads. This could be a bottleneck with respect to transaction execution speed. Nevertheless, in the context of cooperative work, transactions are generally long-lived. Therefore, the extra seconds used for management and control purposes would, in the global picture, be insignificant.

6.6. XML as a specification language

A language that allows runtime interpretation is required in order to allow dynamic specification of transaction models. In practice, this means that a successful modelling language should allow the introduction of new aspects in the transaction model specification while transactions are being scheduled.

Figure 6.7 illustrates the two alternatives of how a transaction model specification can be processed. The most common in existing systems is the first one, where a transaction model is specified, compiled, and then put into execution. The main drawback of this in terms of advanced applications is that if changes are required after transactions have been initiated, they must be suspended or even aborted before new aspects in the specification can be incorporated. Since this may contradict the goal of avoiding unnecessary and expensive repetition of work, the second alternative is preferred – specification and interpretation at runtime. This leads us to an investigation of the use of XML – eXtensible Markup Language (Bray et al. 1998).

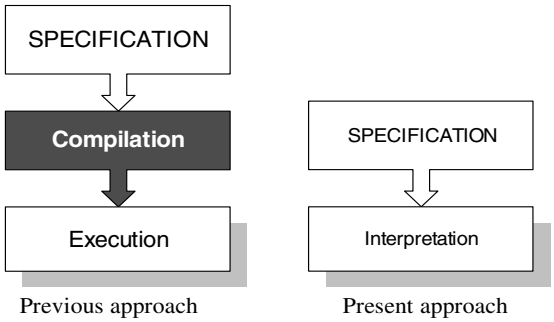


Figure 6.7 Illustration of specification processing alternatives.

Despite the increased use of XML in many applications – including databases – research on the application of XML in the database context seems to be restricted to storing and querying semi-structured data. To our knowledge, XML has not yet been used in connection with transaction modelling.

So, how can XML be utilised in CAGISTrans? Actually, XML and the corresponding DTD – Document Type Definition – in specification of transaction models offers several significant advantages. First, it is easy to develop parsers for XML. In fact, several such parsers are available without cost. Therefore, the effort needed to develop a new specification language was reasonable. Second, XML documents are portable and can be exchanged across different platforms, including different database systems. Regarding openness, this aspect is highly relevant. Third, XML is generic and capable of providing a user-friendly representation of documents. This aspect has been exploited in the representation of the *execution descriptor* in a human readable format, using the Extensible Stylesheet Language (XSL)[1]for transformation of XML to HTML. This has also been useful in monitoring transaction executions. Finally, since XML-documents can be edited and parsed at runtime, it meets one of the main requirements for dynamic respecification of transactional behaviour.

1. See http://www.w3.org/TR/xsl.

Nevertheless, the main concern of XML has been providing effective storage and query mechanisms for the content of XML-documents. This is currently the subject of intensive research. These aspects are beyond the scope of this work. Another concern has been the representation of parallel execution of transactions and conditional rules, as they are not directly supported. This implies that scripting-like languages, such as PHP[1] – a Perl-based scripting language for the Web – are required.

6.7. Concluding remarks

This chapter has presented our CAGISTrans framework for customising and tailoring transaction models to application needs. The main contribution of this work is its way of organising the vital elements of a transaction model to meet the requirements imposed by the dynamic nature of cooperative environments. To achieve this, the framework is built on existing transaction models and transactional frameworks, rather than developing a new transaction model. This is because it is neither practical nor feasible to provide a transaction model that can handle all types of situations. Therefore, extracting positive features from existing models and frameworks can extend these models and frameworks to address the remaining problems within transactional support for cooperative work. The issue of heterogeneity has been addressed through exploiting the middleware principle with respect to supporting advanced transactions independently of the underlying resource systems. This allows the use of the CAGISTrans framework in a wide range of applications, where openness and portability are required.

1. See http://www.php.net.

Chapter 7
Formalising the CAGISTrans Framework

7.1. Introduction

Chapter 6 presented an overview of the CAGISTrans transactional framework. In this chapter we will use this as a starting point to outline the theory, the specification and the design of our CAGISTrans framework. This chapter is organised as follows.

Section 7.2 discusses one way of organising the transaction model elements which aims at meeting the requirements of dynamic and heterogeneous cooperative work environments. To derive these elements, adaptations of the ACID requirements are used as baseline. One of the main findings, discussed in Chapter 6, is the way to separate the model specification into design time and runtime specifications. Section 7.3 shows how transaction specification and execution can be managed at runtime. An important aspect is the use and management of user-defined correctness criteria, suitable for concurrent and dynamic environments. One of the new aspects of our CAGISTrans framework, which existing transactional frameworks have not emphasised, is an integrated support for workspace management. Section 7.4 outlines the way such a support is provided in the framework.

7.2. Allowing design time and runtime specifications

The fact that it is not always possible or feasible to have a complete trans-

action schedule in advance has made it necessary to allow both design time and runtime specifications of transactions. This has implied the necessity of organising the elements of transaction models such that parts of the specification can be done before the actual transactions are executed, while the remaining parts can be defined during runtime. To address this need, our CAGISTrans framework provides transaction modelling building blocks that are organised in *characteristics specification* and *execution specification* (see Figure 6.2).

Fundamental to our framework is the ability for users to tailor transaction properties to meet the requirements of their applications. For this reason we have advocated the need for specifying the "ACID requirements" to be applied – i.e., customised non-ACID properties. The following sections explain how this specification can be accomplished.

7.2.1. Customising the ACID properties and its implications on transaction characteristics

This section shows that customisation of the ACID properties may well be used as bases for the transaction characteristics specification.

To deal with the undue strictness of full ACIDity, the ability is required to seamlessly tailor the ACID requirements. As some situations – within a single advanced application – may still demand parts of the ACID requirements to be maintained, most situations may see them just as a burden. Thus the proposed solution is to allow users to customise the ACID properties according to the needs of their applications, rather than providing them with fixed properties. Basically, the idea is to allow users to first identify the main requirements, and thereafter to designate the appropriate properties.

Customisation of the ACID properties should only affect the atomicity and isolation requirements. However, consistency and durability should be maintained by definition. This is despite of the fact that some transactions may have to tolerate some temporary inconsistencies, and parts of a transaction execution may not be required to be permanent.

7.2.2. Preserving consistency and durability

As the fundamental requirements underlying transaction processing are that (1) executing transactions always transform a database from one consistent state to another consistent state, and (2) once the execution is terminated the final result is permanent (Gray and Reuter 1993). These requirements are necessary both to ensure the correctness of data shared among users and to make sure that all committed results survive possible crashes. These have been the

main motivation for the need to preserve both the consistency and durability properties.

To recapitulate, full consistency implies that a set of user-defined or database dependent criteria must be provided. In addition, policies determining relevant mechanisms and specifying the rules for their applicability are necessary (Ramampiaro et al. 1999, Ramampiaro and Nygård 2001).

Further, full durability implies the use of logging facilities that maintain all necessary information about executing transactions in a persistent store. In addition, there is a need for mechanisms that allow transactions to explicitly discard the effect of committed results – e.g., by transaction compensation (Korth et al. 1990, Levy et al. 1991).

7.2.3. Switching between full and relaxed atomicity

Existing transaction models for advanced applications have emphasised the necessity of relaxing the atomicity property to meet the requirements of long, interactive transactions. The suggested solution is often to provide abort management with finer granularity than that provided by full atomic transactions – i.e., allowing partial aborts. However, there are still situations where full atomicity is necessary to achieve acceptable processing. An example is Case 5 in Section 5.4. Ideally, users should be provided with the ability to chose the way to manage transaction aborts. This is why the necessity for seamlessly switching between full and relaxed atomicity is stressed. Thus, the flexibility is increased, in terms of the ability to manage transaction termination exactly according to the application needs.

The necessity of specifying the transaction structure, distinguishing between *flat* and *nested structures* has already been indicated. In this way, transaction abortion may be managed in accordance with the actual structure. Assuming that transactions that are short-lived and do not involve interaction, are flat. They are seen as atomic satisfying the all-or-nothing law.

By contrast, assuming that long-lived transactions are nested, consisting of recursively defined constituent transactions, aborts of transactions can be managed in a more controlled manner – cf., Moss' nested transaction model (Moss 1982).

It is now proposed to specify transaction atomicity by means of transaction dependencies. This is based on the dependency theory from e.g., (Chrysanthis and Ramamritham 1994, Georgakopoulos et al. 1996). Note that existing solutions involving dependencies assume the existence of parent to children abort dependency schemes, specifying by default that if a parent

transaction aborts then all its children automatically abort as well. The abortion of a child, on the other hand, does not have any direct effect on the parent. The source of dependencies in such a case is thus the transaction structure.

The main disadvantage of such an approach is that abortion of a parent always discards the effects of its children, making it impossible to define alternative, more relaxed, abortion schemes. For example, consider a software project consisting of several activities – as in our Aviasoft example. Although the project is cancelled, it might be desirable to "save" the results of some already completed activities that could be useful in future projects. Moreover, there are examples where it is necessary to define the effects of a child's abortion on its parent as well as on possible siblings. This stresses the necessity for defining a more generalised dependency scheme than that based on structure. This is called an *atomicity abort dependency scheme.*

The idea is to allow an explicit designation of a desired abort specification depending on the application needs. In this sense, prevailing abort dependencies can be defined regardless of any existing access dependencies – i.e., read/write dependencies – or their structural dependencies – e.g., parent/children dependencies. The specification of these dependencies is primarily intended to be done for specific executions of transactions.

Consider a transaction T with constituent transactions $t_0, t_1, t_2, \ldots, t_n$ – i.e., $T = \{t_0, t_1, t_2, \ldots, t_n\}$. To allow the transaction scheduler to deduce which transactions have to abort in case a transaction t_i fails, it manages sets, containing lists of affected transactions, called *AbortSet(t_i):*

AbortSet(t_i) is the set of transactions that have to abort if t_i aborts*:*

$$AbortSet(t_i) = \left\lfloor t_j \in T \,\middle|\, abort(t_i) \rightarrow abort(t_j), i \neq j \right\rfloor_{\bullet} T = \left\lfloor t_0, t_1, t_3, \ldots, t_n \right\rfloor_{\bullet}$$

This means that *AbortSet(t_i)* is a set of transactions t_j such that if t_i aborts, t_j must also abort.

In the CAGISTrans framework, there is a data structure maintaining a set of *AbortSet(t_i)* that the transaction manager uses to determine which other transactions must abort in case a specific transaction aborts. This means that when a transaction t_i aborts, the scheduler will execute `DoAbort(t`$_i$`)` as follows:

(1) Mark t_i "aborted".

(2) If $AbortSet(t_i) \neq \emptyset$ then for each $t_j \in AbortSet(t_i)$
if t_j is not already aborted then `DoAbort(t`$_j$`)` .

To illustrate, Figure 7.1 depicts a nested transaction T_1 with subtransactions $T_{1.1}$, $T_{1.2}$ and $T_{1.2.1}$. From this, we get the constituent transactions $T = \{T_1, T_{1.1}, T_{1.2}\}$ and $T' = \{T_{1.2}, T_{1.2.1}\}$. The arrows denote the specified abort dependencies among the involved transactions – i.e., $T_{1.1}$ depends on T_1, and T_1 and $T_{1.2.1}$ depend on $T_{1.2}$. Hence, following our definition, $AbortSet(T_1) = \{T_{1.1}\}$, $AbortSet(T_{1.1}) = \{\}$, $AbortSet(T_{1.2}) = \{T_1, T_{1.2.1}\}$ and $AbortSet(T_{1.2.1}) = \{\}$.

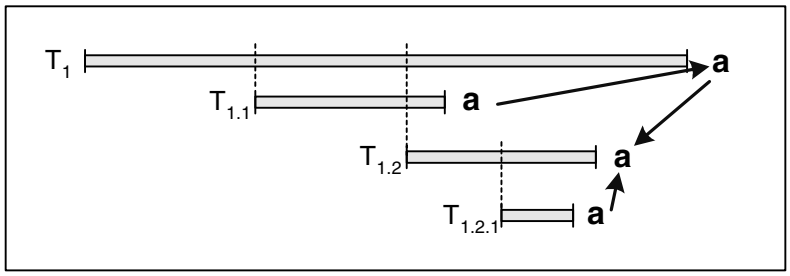


Figure 7.1 Illustration of abort dependency.

As this indicates, if T_1 aborts then $T_{1.1}$ must also abort. The abortion of $T_{1.2}$ will, cause T_1 and $T_{1.2.1}$ to abort. In other words, the abortion of $T_{1.2}$ will affect its parent transaction T_1 as well as its child $T_{1.2.1}$. But the abortion of $T_{1.1}$ or $T_{1.2.1}$ will not affect other (sub) transactions.

How does all this affect the execution of transactions? We have assumed that all transactions with relaxed atomicity are nested.The effect of the specifications of the atomicity properties are relevant in the way transactions realise their specified structures during runtime. This means that some transactions may have to restructure – i.e., spawn new transactions and delegate some responsibilities, making it necessary to provide operations for dynamic restructuring (see Section 7.3 for an elaboration).

7.2.4. Switching between full and relaxed isolation

Traditional multi-user database systems often assume that users do not need to cooperate on data, only compete, giving them the impression of being solely in charge of some specific resources. Clearly, such an assumption contradicts the cooperative work philosophy. In fact, sharing of tentative data is needed for cooperation to be possible. Therefore, transactions must be able to reveal their intermediate results. However, this may again cause undue cascading abortion, which was originally one of the main reasons for the isolation requirement (see Section 2.2). Hence, it is important to have an acceptable abortion scheme to ensure that only those that are directly affected by a failure have to abort. The other cooperating transactions should be able to proceed as normal. To achieve this, we again use the *AbortSet* and its corresponding algorithm from above, but now we are also interested in dependencies among

transactions that are not "family" related. Thus, instead of only traversing the *AbortSet* containing children of a specific transaction, the abortion algorithm will also go through an "abort set" containing cooperating transactions that should be aborted with a specific transaction.

Formally, we now define a constituent transaction set T_i as $T_i=\{t_{i.0},t_{i.1},t_{i.2},...t_{i.Ni}\}$, $i=1,..., m$. Thus, the new *AbortSet* is defined as

$$AbortSet(t_{i.j}) = \left\lceil \left\lceil t \in \bigcup_{i=1}^{m} T_i \,\middle|\, abort(t_{i.j}) \rightarrow abort(t) \right\rceil \right\rfloor$$

where $T_i = \{t_{i.j}, t_{i.j}, t_{i.j}, ..., t_{i.\mathrm{Ni}}\}$, $i = 1, 2, ..., \mathrm{m}$.

This means that the abort set of a transaction $t_{i.j} \in T_i$ may contain transactions that are children of other transactions.

To illustrate this, consider the two nested transactions T_1 and T_2 depicted in Figure 7.2. Using our definition, $AbortSet(T_1) = \{T_{1.1}\}$, $AbortSet(T_{1.1}) = \{\ \}$, $AbortSet(T_{1.2}) = \{T_1, T_{1.2.1}\}$, $AbortSet(T_{1.2.1}) = \{T_{2.2}\}$, $AbortSet(T_2) = \{T_{2.1}, T_{2.2}\}$ and $AbortSet(T_{2.1}) = AbortSet(T_{2.2}) = \{\ \}$. This implies that an abortion of $T_{1.2.1}$ will cause $T_{2.2}$ to abort, while $T_{2.1}$ may proceed as normal.

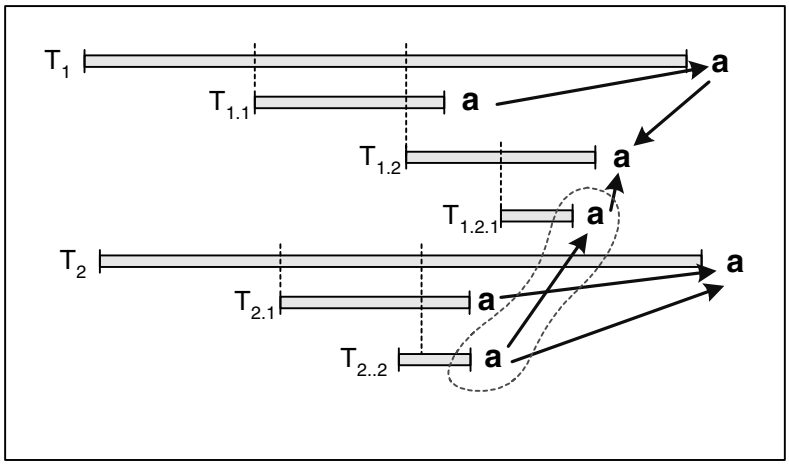


Figure 7.2 Illustration of abort dependency across two nested transactions.

In addition to this relaxed abortion scheme, it is necessary to have relaxed correctness criteria allowing the above mentioned sharing of intermediate results. Of course, such criteria will have to be user defined since not all applications have the same needs. How such a criterion is provided in the CAGISTrans framework is discussed in Section 7.3.3.

Further, to achieve a specified criterion, we have to define which policy should be applied. Such policies are necessary to determine the mechanisms that are relevant and the rules for when and how to use the mechanisms. Table 7.1 provides an overview of the correctness criteria and policies associated with the isolation property.

		Full Isolation	Relaxed Isolation
Correctness Criteria		Serialisability	User defined, see Section 7.3.3
Applied Policies	Mechanisms	Strict locking and awareness	Flexible locking, workspaces and awareness
	Rules	2PL protocol	User-controlled lock protocol and Collaborative lock protocol

Table 7.1 Isolation with corresponding correctness criteria and applied policies.

For full isolation, we apply the serialisability correctness criterion and use the standard locking policies described in the literature (Bernstein et al. 1987). For relaxed isolation, on the other hand, we apply user-defined correctness criteria, which are elaborated on in Section 7.3. The relevant mechanisms are flexible locking (treated in Section 7.2.4.1) – *user-controlled locks* and *collaborative locks*, workspace usage (treated in Section 7.4), and awareness mechanisms (treated in Section 7.2.4.2).

7.2.4.1. Flexible locks

The *user-controlled lock* protocol is defined and implemented as follows:

(1) Locks are acquired to lock specific objects – i.e., each object has an attribute telling which transaction is holding a lock on it.

(2) No lock mode – e.g., read-lock vs. write-lock – is required, as the lock is associated directly with an object rather than with an operation.

(3) Once an object is locked, no other transactions may acquire a lock on that object until the existing lock is released.

The main characteristic of this locking protocol is that locking is basically done interactively, without following any strict automatic locking rules. Each lock is accompanied by a notification tag providing information about the owner of the lock and the object being locked. In this way, when a lock request is issued, the system will be able to provide the necessary feedback about the current lock held (see also Section 7.2.4.2). Such feedback would help the requester finding appropriate course of actions upon conflicts. There are several possibilities. First, normally if the object is locked by a transaction, the requesting transaction would have to wait until the lock on this object is released.

However, since (long) waits are often undesirable, alternative actions could be useful. Note that when a transaction request a lock, the lock holder will be made aware of the request by default. He/she may then

(1) "set a *permit*" on the requesting transaction, and downgrade the lock into collaborative lock (see below), or

(2) if the transaction no longer needs to use the data, it can release the lock so that the lock request can be granted.

In cooperative activities, two or more transactions may have to cooperate on the same object. The locking protocol above restrict explicit cooperation. To cope with this, we apply the *collaborative lock* protocol. This is defined and implemented as follows:

(1) As with user-controlled locks, these locks are acquired to lock specific objects.

(2) And again, no lock modes are required.

(3) But, although an object is locked, other transactions may also acquire a lock on that object, regardless of the operation semantics.

(4) When a transaction requests a lock on an object already locked by another transaction, it will be informed about the locking situation, and asked to specify the intention: *browse*, *incorporate*, or *modify*.

   (a) If the intention is to *browse*, the lock is granted right away.

   (b) If the intention is to *incorporate,* the lock is granted, but the sequence of future actions must be specified using *demands* (See Section 7.3.3).

   (c) If the intention is to *modify*, the lock owner and the requester will first be warned. Thereafter, they are required to solve the conflicts through negotiation.

As can be inferred, this locking protocol allows collaborating transactions to hold conflicting locks. These are similar to those found in the Solaris[1] file system. However, the main difference is that before a lock request is granted, the intention must be specified, and concurrent updates are allowed only after explicit agreement among the collaborative partners.

The premises for the use of collaborative locks in the CAGISTrans framework are that (1) there is not always a strong link between what objects that

1. Solaris is a trade-mark of Sun Microsystem. See http://www.sun.com.

transactions read and what actions that they perform based these reads, and (2) in some situations, inconsistency can be allowed for limited periods of time – i.e., until a time the users choose to resolve any induced conflicts.

In this view, the lock intention proposed here can be used to overcome conflicts in the following manners:

- *Browse lock intention* allows a transaction to read objects that are being locked by other transactions. However, it does not allow the reading transaction to manipulate – e.g., deleting or modifying – the objects that are read. Further, it is assumed that no explicit changes of actions are required due to changes of the objects read (see the discussion concerning the example in Section 7.3.2). Note that the users will be notified of any changes to the objects read. Thus, if changes to other objects are considered useful, the lock intention on these objects should be upgraded to *incorporate*. A browse lock intention is by default compatible with browse, incorporate and modify lock intentions.

- *Incorporate lock intention* allows a transaction to conditionally read objects that are being locked by other transactions. As for *browse,* it does not allow the reading transaction to manipulate – e.g., deleting or modifying – the objects that are read. However, it assumes explicit changes of actions when updates on the objects read occur. This means that the actions must conform to the updates on the object being incorporated. Hence, *demands* should be specified to determine necessary actions. Such actions will, for instance, "compensate" the changes made by factors such as a new "incorporation" (see the illustrations in Case 4, Section 7.3.3.2). An incorporate lock intention is by default compatible with browse and incorporate intentions, but not with modify intentions. However, this incompatibility can be overridden through the use of demands.

- *Modify lock intention* allows a transaction to manipulate objects (conditionally if the objects are being locked by other transactions). Conflicting *modify* lock intentions lead to negotiation in order to resolve the incurred conflicts. Such negotiation is performed on the basis of the object that is being altered. The involved users that hold conflicting locks on that object are informed about the conflict. Based on this, they may decide upon agreement on the order of their updates, and whether the lock requesting transactions have to cancel the lock acquisition. Note that other related "incorporation" would have to specify demands in accordance with the resulting modifications. A modify lock intention is, by default, incompatible with incorporate and modify intentions, but

this incompatibility can be overridden through negotiation, and demands may have to be specified. A modify intention is compatible with browse.

To summarise this discussion, Table 7.2 shows the compatibility matrix of the lock intentions.

	Browse	*Incorporate*	*Modify*
Browse	True	True	True
Incorporate	True	True	False[a]
Modify	True	False[a]	False[b]

Table 7.2 Compatibility matrix of the lock intentions.

a. Overridable, but demands must be specified if overridden.
b. Overridable, but negotiation is required.

This discussion shows that the locking protocol relies on users choosing appropriate collaborative lock intentions. It is recognised that this could impose some important challenges that still deserve careful investigation. In particular, indirect conflicts can occur, which may again lead to concurrency-induced inconsistency. Therefore, it is necessary for the CAGISTrans framework to provide sufficient user support in order to address this issue. However, it is important to still consider the trade-off between (1) total system control, resulting in error-free interactions and (2) flexibility, allowing users to choose appropriate actions according to their needs. Early studies of computer supported teamwork have indicated that humans are indeed prepared to accept some responsibility for processing information (Ellis et al. 1991). Nevertheless, it is recognised that it is still of immense importance that the system offers appropriate support in order for the users to be able to handle inconsistency and concurrent conflicting access to objects. Note that in providing such support one should not neglect the importance of finding the sensible trade-off between *supporting* teamwork and *controlling* it (Sommerville and Rodden 1993).

Also note that, as mentioned earlier, user-controlled locks may be degraded to collaborative locks upon request. In other words, if an object is locked using a user-controlled lock, this lock can be degraded to a collaborative lock when another transaction requests to access the object. However, before such access can be accomplished, a *permit* relationship must be established (again see Section 7.3.3).

7.2.4.2. Awareness mechanisms

Awareness in CAGISTrans is realised with event notification mechanisms. They are aimed at providing users engaged in an activity with knowledge about events that are of possible interest to them.

There are several types of events that are relevant. Related to concurrent execution of transactions, significant events may be associated with transaction start, transaction termination, acquisition of locks and object change. These are realised as follows:

1. *Notify on begin.* When a transaction starts executing, a notification message is broadcasted to all involved parties.
2. *Notify on terminate.* When a transaction commits or aborts, a notification message is sent to all involved parties.
3. *Notify on lock.* When a lock is acquired or released by a transaction, a notification message is broadcasted to all involved parties, specifying which type of lock was acquired or released and on which data the lock is/was applied.
4. *Notify on change.* When a data object is being altered by a specific transaction operation, a notification message is sent to all involved parties.

Here, involved parties mean all users executing transactions that share a common resource. For example, in cases where workspaces apply, all users connected to a shared workspace can be seen as involved parties.

With relaxed isolation, all four notification events are useful due to potential cooperation. With full isolation, on the other hand, the knowledge about the existence of other transactions is normally less crucial. Therefore, none of the notification events are primarily required. Still, events (1) and (2) may be useful. Recall, for instance, the scenario from Case 5 in Section 5.4. Although the execution of an agenda update must be atomic and isolated, affected engineers would appreciate notification about the beginning and the termination of this update, so that they may accommodate their activities accordingly.

Summarising (see Table 7.3), notification types (1), (2), (3) and (4) are relevant when the isolation property is relaxed, whereas types (1) and (2) might be useful when the isolation property is full.

	Full Isolation	Relaxed Isolation
Notify on begin	Useful	Relevant
Notify on terminate	Useful	Relevant
Notify on lock	N/A	Relevant
Notify on change	N/A	Relevant

Table 7.3 Relevance for notification types.

7.2.5. Analysing the combination of isolation and atomicity properties

Table 7.4 indicates possible combinations of atomicity and isolation properties of transactions. Combining full atomicity and full isolation is readily okay as for the ACID models – cf. (Härder and Reuter 1983).

	Full Atomicity	Relaxed Atomicity
Full Isolation	Okay	Okay
Relaxed Isolation	Not always convenient	Okay

Table 7.4 Possible combinations of atomicity and isolation properties.

We may also combine full isolation with relaxed atomicity, allowing transactions to provide partial rollback or controlled abort management, but at the same time prohibiting cooperation. An example of a transaction model having this combination is the nested transaction model (Moss 1982).

However, combining full atomicity and relaxed isolation may not always be convenient due to the cost of rollback. An example is a variant of open nested transactions (Weikum and Schek 1992) allowing subtransactions to cooperate. If a transaction aborts then all associated transactions must abort too, whether they have performed conflicting operations or not. Clearly, such a requirement would not be suitable within cooperative environments involving several associated interactive activities of long duration. Still, this combination could be useful for small cooperative tasks – i.e., tasks that do not involve too much invested effort – that must still meet the all-or-nothing objective for the results to be reliable.

The final case – combining of relaxed isolation and relaxed atomicity – will always be okay, assuming the consistency and durability requirements can

be met. This last combination is one of the main issues that will be elaborated on in the rest of this thesis.

7.3. Management of transactional behaviour at runtime

Adaptability is a prerequisite for the support of dynamic cooperative environments. Adaptability means the ability to tailor the provided support to different needs, including those that may change while involved activities are in progress. In the context of transactional support, meeting such a requirement is not a trivial matter. The main challenges are concerned with how to apply changes while transactions are being processed.

However, it may be concluded from the discussion in Section 6.4.1 that we can cope with these challenges by allowing a user to specify transaction models partially at design time and partially at runtime. As was pointed out, three main building blocks are necessary to manage the transactional behaviour during runtime:

- *Operations* used to manage the execution of transactions
- *Advanced operations* specifying the actions a transaction can execute
- *Rules* defining constraints to manage and control the effect of transaction executions

These three issues will be covered in Sections 7.3.1, 7.3.2 and 7.3.3 respectively.

7.3.1. Management operations

Management of transaction execution is achieved through operations that manage initiation, termination and restructuring respectively.

7.3.1.1. Managing initiation and termination

We can assume that every transaction starts executing with a *begin* and eventually terminates with either a *commit* – if it succeeds, or an *abort* – if it has to discard any changes. To allow the widest possible application areas, CAGISTrans is designed to enable both automatic management – i.e., transparent initiation and termination applicable to ACID transactions, and interactive management – i.e., user-controlled initiation and termination relevant for cooperative interactive environments. The former is realised by allowing the system to perform the initiation/termination without any intervention by the user.

The latter, on the other hand, is realised in such a way that a user can freely choose the way transactions execute and terminate. However, to avoid anarchy, there is a need to define dependencies among the involved transactions. These will indicate the *order* that transactions begin and commit, and the *effect* of a transaction's abort on other transactions. This means that the freedom of users to execute and terminate transactions will be restricted by the prevailing *begin dependencies* (Chrysanthis and Ramamritham 1994), *commit dependencies* (Chrysanthis and Ramamritham 1994) and *abort schemes*.

For example, initiation of a transaction T_i may not be accomplished if a begin dependency specification says that another T_j has to start first. Similarly, a specific transaction T_i may not be terminated if there is a commit dependency specification that enforces T_i to wait for another transaction T_j to finish. Furthermore, aborting a specific transaction T_i may cause another transaction to abort depending on the prevailing abort scheme (see Section 7.2.1).

Let us formalise these rules adopting Klein's event rules (Klein 1991). Let e be an event. The order in which two consecutive events e_1 and e_2 must appear is denoted by $e_1 < e_2$.

Using this, the above dependency rules can be defined as follows:

- *Begin dependency* (Chrysanthis and Ramamritham 1994): If a transaction t_2 is begin dependent on t_1, then t_2 may only begin after t_1 has started. This can be expressed as $begin(t_1) < begin(t_2)$.

- *Commit dependency* (Chrysanthis and Ramamritham 1994): If a transaction t_2 is commit dependent on t_1, then t_2 may only commit after t_1 has started. This can be expressed as $commit(t_1) < commit(t_2)$.

To illustrate the utility of these dependencies, let us consider our Aviasoft scenario from Section 5.1.

Let t_1 be a coding task for the GUI module and t_2 be a quality assurance (code inspection) task for the same module. Clearly, t_2 cannot start until t_1 has started and has made available any code for inspection. Until then, t_2 has to wait. Note that normally this would not require t_1 to commit first. Similarly, we could say that the coding task is not finished until the assurance task is accomplished. This means that the commitment of t_2 must occur before t_1's commit.

7.3.1.2. Managing dynamic restructuring

The ability of a transaction to delegate responsibilities and spawn new transactions during runtime is referred to as dynamic restructuring. Referring

to our discussion in Section 6.4.1.2, this has been identified as a useful cooperation primitive in addition to being a means to realise a specified structure during runtime.

The main reason for adopting the basic ideas of this notion is the ability to support open-ended activities. By allowing a transaction to delegate responsibilities, it will be able to release resources that are no longer needed until it terminates. In this way, resources can be made available to other transactions at an earlier stage. For example, although a transaction aborts, some of its resources may still be used by other still executing transactions.

In CAGISTrans we speak of two types of responsibilities: object locks and operations. A transaction may thus transfer locks on specific objects to another transaction, giving it the responsibility to – e.g., accomplish updates on these objects. It may also transfer operations to another transaction, giving it the responsibility to commit these operations. From this perspective, the difference between our approach and that of the Split and Join transaction model is the use of both operation and object lock delegation. Restructuring in the Split and Join transaction model is based on object lock transfers only. This is in addition to the fact that transactions in that model stick to serialisability, but our framework allows user-defined correctness criteria.

Performing delegations

Our dynamic restructuring will depend on the specified structure (see Table 6.6). Flat transactions do not spawn new transactions, but may delegate responsibilities to other transactions. This requires that a transaction maintains two sets containing, respectively, object locks and the operations for which it needs to delegate responsibilities. We call these `ObjSet` and `OpSet`, respectively. An operation `delegate(`t_i`, `t_j`, ObjSet, OpSet)` performs a resource transfer from a transaction t_i to another transaction t_j. This means that when a transaction t_i executes `delegate`, this transaction will release its locks on all objects in `ObjSet`, and t_j will immediately take over these locks. In addition, all operations in `OpSet` will be removed from t_i's history, before they are transferred to t_j's history. Consequently, at the time t_i terminates, the operations that have been delegated to other transactions will not be affected.

Unlike flat transactions, nested transactions may spawn new transactions. Thus dynamic restructuring may involve the generation of constituent transactions. In that case, `delegate(`t_i`, `t_j`, ObjSet, OpSet)` transfers responsibility from t_i to one of its subtransactions t_j. At the time this operation is issued, t_j may be active or not. If t_j is active, then the restructuring is achieved as above. If it is not active, it will be initiated as a child of t_i, before the resource transfer is accomplished as before.

From the above discussion it can be inferred that from the delegation point of view, delegations for flat transactions appear more general than those for nested transactions. The main reason for this is to keep the delegation complexity at a manageable level. This would ease the decision for finding the delegatee. Without such restriction, delegation across the nested structure might be relevant, which could imply that any subtransaction from a nested tree (subtree) could, in theory, perform the delegation to another subtransactions in another tree (or subtree). As stated earlier, however, one of the main reasons for having delegation in the CAGISTrans framework is primarily to allow dynamic restructuring. Therefore, its management can be kept as simple as possible by emphasising a more generalised delegation for flat transactions, and focusing the restructuring for their nested counterparts.

Recovery issues

It is a prerequisite that delegation of responsibilities is atomic to ensure consistent "results". This means that when issuing `delegate(`t_i`, `t_j`, ObjSet, OpSet)`, either all locks on objects in `ObjSet` and all operations in `OpSet` are successfully transferred from t_i to t_j or none.

Information concerning locks on objects in `ObjSet` must be recoverable, whereas it is not necessary to make an explicit recovery management on the information concerning `OpSet` since it will still be saved in the delegator's history until the delegation is accomplished.

7.3.2. Advanced operations

Advanced operations mean commands specified at an abstraction level higher than, but based on, *read* and *write* operations. As pointed out in Section 6.4.1, there are several advantages in using advanced operations in CAGISTrans. An important one is that advanced operations allow us to exploit semantic knowledge about an operation. An example of such semantics is the return information from executed operations – e.g., the changes operations have made. Another example is the intention of an operation. Here, for instance, we can speak of two types of read intentions; *browsing* and *incorporation.* Browsing means that even if an object being read by a user had a different value, this would not cause the user to perform different actions. In other words, with browsing, changes will not affect users' future actions. On the other hand, if the intention is incorporation, changes on an object read are likely to affect the way users act.

How do we realise advanced operations with intentions? We define a general advanced operation as a tuple $Op = \langle Optype, InSet, OutSet, Intent \rangle$, where *Optype* denotes the type of operation, *InSet* is the set of input objects –

i.e., objects read, *OutSet* is the set of output objects – i.e., objects written, and *Intent* indicates the intentions of each input argument.

Using this, we can specify a general conflict rule based on operation commutativity (Weihl 1988); two operations are in conflict if the order in which they are executed matters. Disregarding intentions, this means that two operations $Op_1 = \langle Optype_1, InSet_1, OutSet_1 \rangle$ and $Op_2 = \langle Optype_2, InSet_2, OutSet_2 \rangle$ commute if the objects read by Op_1 are different from the objects written by Op_2, the objects read by Op_2 are different from the objects written by Op_1, and *Op1* and *Op2* do not update any common object:

$$(InSet_1 \cap OutSet_2 = \emptyset) \wedge (InSet_2 \cap OutSet_1 = \emptyset) \wedge (OutSet_1 \cap OutSet_2 = \emptyset)$$

With intention knowledge, we relax this commutativity rule such that even if the intersection between an *InSet* and *OutSet* pair is not empty, the two operations will still commute when the objects in the intersection are for browse with the inputting transaction.

In practice, we substitute the *Intent* in the operation tuple with a *BrowseSet*, assuming that all objects found in *InSet* but not in *BrowseSet* are by default incorporated. Based on this, the new commutativity rule can be formally defined as follows. Considering $Op_i = \langle OpType_i, InSet_i, OutSet_i, BroweSet_i \rangle$:

$$Compatible(Op_i, Op_j) \Leftrightarrow (OutSet_i \cap OutSet_j = \emptyset) \wedge ((\forall Ob \in (InSet_i \cap OutSet_j) Ob \in BrowseSet_i) \vee (\forall Ob \in (InSet_j \cap OutSet_i) Ob \in BrowseSet_j))$$

This means that two operations Op_i and Op_j are compatible iff (1) Op_i and Op_j do not perform updates on any common objects, and (2) objects read by one of the operations and updated by the other operation only are for browse with the inputting transaction. Note, this assumes that all elements of *BrowseSet*$_i$ are elements of *InSet*$_i$; $\forall Ob, (Ob \in BrowseSet_i) \rightarrow (Ob \in InSet_i)$.

To illustrate, suppose that Aviasoft's software development environment provides the commands *Edit_interface*, *Edit_class*, and *Compile_class* for a coding process. Assume that all these are abstractions of read and write operations. Consider that two modules are to be created as part of the coding process as illustrated in Figure 7.3 – i.e., the graphical user interface module (*GUI* for short) and the process module (P) processing inputs from the *GUI* and producing data to be displayed by the *GUI*. Further, *Edit_class(GUI)* implements the interface part of the process module, called process interface (*P_i* for short). As indicated in the illustration, this means that changes on *P_i* will affect the execution

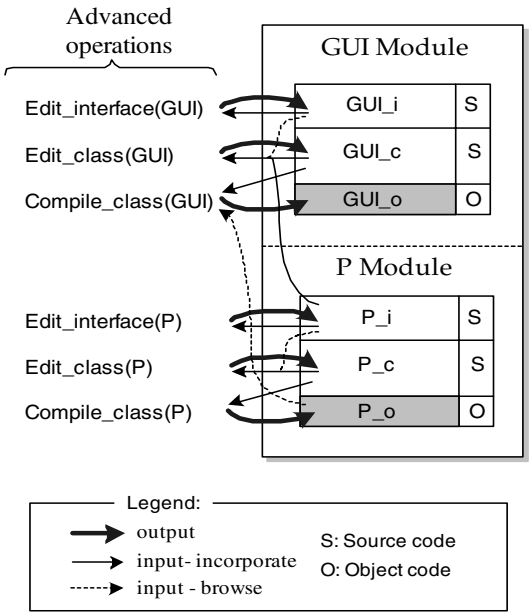


Figure 7.3 Advanced operations illustration.

of *Edit_class(GUI)*. But, as depicted, although *Edit_class(GUI)* reads *GUI_i*, changes on this object will not directly affect that operation, in that it had, for example, to make new changes on GUI_c. To produce GUI_c, it suffices to include GUI_i in its "import list". It is similar for *Edit_class(P)* and *Compile_class(P)*.

Here, we can say that a read does not have a direct effect on an operation if there is no explicit connection between what objects and the values that the transaction reads and what it performs as a consequence. This means that even if the value for an object read by a transaction was modified, it would not have to perform different actions, as for example to re-update other objects. As such, some browse reads could well be executed as transactions on their own. However, sometimes their execution with other "conflicting" operations may still be useful. For example, let us consider the `Edit_class(P)` operation in Figure 7.3. To produce/update the class code `P_c`, both `P_c` and `P_i` are read. If `P_c` already exists, its previous "value" is incorporated for updates. However, `P_i` is required here only for the user running `Edit_class(P)` to know what has been made on `P_i` to find out whether the "list" of methods provided in it conforms to those implemented in `P_c`. P_i is in addition to this needed for compilation purposes. Note that although the "list" provided in P_i is still incomplete and is being changed, the user would still be able to produce `P_c`, and modification of P_c is not required. From this it can be concluded that any inconsistency between P_i and P_c does not affect the production of P_c, per se.

Using the advanced operation type, we get the specifications given in

Table 7.5.

Operations applied on GUI
Edit_interface(GUI) = ⟨Edit_interface, {GUI_i}, {GUI_i},{ }⟩
Edit_class(GUI) = ⟨Edit_class, {GUI_c, P_i, GUI_i}, {GUI_c}, {GUI_i}⟩
Compile_class(GUI) = ⟨Compile_class, {GUI_c, P_o}, {GUI_o}, {P_o}⟩
Operations applied on P
Edit_interface(P) = ⟨Edit_interface, {P_i}, {P_i}, { }⟩
Edit_class(P) = ⟨Edit_class, {P_c, P_i}, {P_c}, {P_i}⟩
Compile_class(P) = ⟨Compile_class, {P_c}, {P_o}, { }⟩

Table 7.5 Illustrations of advanced operation specification.

Applying our advanced compatibility rule, we get for example:

¬*Compatible*(*Compile_class(GUI), Edit_class(GUI)*)
¬*Compatible*(*Compile_class(P), Edit_class(P)*)
¬*Compatible*(*Edit_class(GUI), Edit_interface(P)*)
Compatible(*Edit_class(GUI), Edit_interface(GUI)*)
Compatible(*Edit_class(P), Edit_interface(P)*)
Compatible(*Compile_class(GUI)*,*Compile_class(P)*)

These results will be used in the illustrative examples in the rest of this chapter.

7.3.3. Managing and controlling transactional behaviour

Transaction executions must be managed in a controlled manner to achieve acceptable data consistency. Traditionally, serialisability has been the ultimate and widely accepted criterion to achieve data consistency (Bernstein et al. 1987). However, as pointed out earlier, a serialisability criterion requires transactions to be isolated, thus prohibiting cooperation. To overcome such a restriction, a set of user-controlled correctness constraints has been included to allow a controlled sharing of data, and ensure that transactions terminate with consistent final results. Such constraints may be used when the serialisability criterion is considered inappropriate.

7.3.3.1. General discussion

To recapitulate, the constraints in CAGISTrans explicitly specify transaction interleavings that are (1) *prohibited*, (2) *allowed*, and (3) *mandatory*. Constraints (1) are realised using *conflicts*, which define operations that cannot execute concurrently to hinder incorrect effects. Constraints (2) are realised using *permits*, which identify operations that in general are considered conflicting

by (1), but that still can execute concurrently. Finally, constraints (3) are realised with *demands*, which specify operation sequences that must appear to achieve correct executions. Different combination of conflicts, permits, and demands thus constitute the user-defined correctness criteria in CAGISTrans.

Figure 7.4 is a general illustration of how combinations of the three constraint types can be exploited as user-defined correctness criteria. A table containing a set of prohibited interleavings specifies the *conflicts.* This consists of a list of operation pairs that cannot be executed concurrently, for the currently executing transaction set. Consider now that T_1={t_5, t_6, t_{10}} and T_2={t_3, t_7, t_4}, are two sets of cooperating transactions. While interacting, users running transactions in T_1 and T_2 find out that they want to share some specific objects. However, due to the specified *conflicts*, this is impossible. There are, in particular, two specific pairs of operations that they would like to execute concurrently. To cope with this, they must specify a *permit* relationship for each transaction set. However, the two permitted interleavings may now introduce some concurrency anomalies. Therefore, they might also have to specify *demand* rules, determining sequences of operations that the transactions in T_1 and T_2 have to execute. Note that such *demanded* interleavings must still obey the conflicts defined for the two transaction sets.

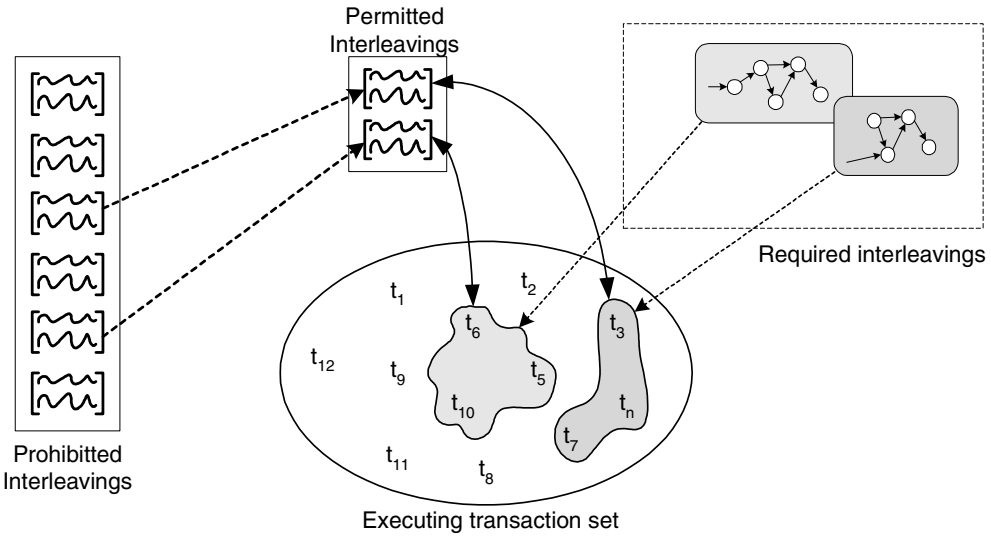


Figure 7.4 Illustration of the use of the three constraints.

7.3.3.2. The necessity of conflicts, permits and demands

This section discusses the necessity of diverse combinations of the three constraint types.

Case 1: Conflicts and Permits only

This case assumes that we only have conflicts and permits, where the permits allow some specific transactions to violate some of the conflict rules. An analogous example is allowing a transaction t_j to write an object read by another uncommitted transaction t_i. As long as t_i's read does not affect its future computation, any changes to this object will not be critical. But if this is not the case, problems might arise. This means that there are situations where conflicts and permits alone would not be sufficient to guarantee correctness. Rules specifying what sequences of actions must be executed for the final results to be acceptable from the viewpoint of both t_i and t_j are also needed.

To illustrate, two engineers Tom and John are assigned the responsibility to create the two modules *GUI* and *P* from Section 7.3.2. Figure 7.5 illustrates a possible interaction scenario.

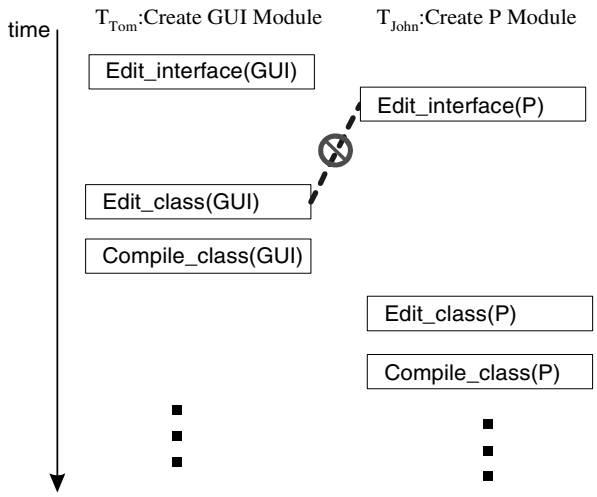


Figure 7.5 Incorrect scenario.

According to our commutativity (conflict) rule, this scenario is illegal because of the concurrent execution of *Edit_interface(P)* and *Edit_class(GUI)*. As a result, Tom must wait until John finishes. Assume, however, that Tom and John find such a wait unacceptable. Hence, they need a permit relationship to enable their interaction. We may define this as *permit*(T_{Tom}, T_{John}, [*Edit_class, Edit_interface*], [*GUI, P*]). This allows Tom and John to execute *Edit_class* and *Edit_interface* upon *GUI* and *P*, even if this violates a commutativity conflict. Taking this permit relationship into account, Figure 7.6 shows the scenario that may be considered correct.

Note, to achieve recoverability (Bernstein et al. 1987), we must ensure that T_{Tom} does not commit before T_{John}, specified as a commit dependency – i.e.,

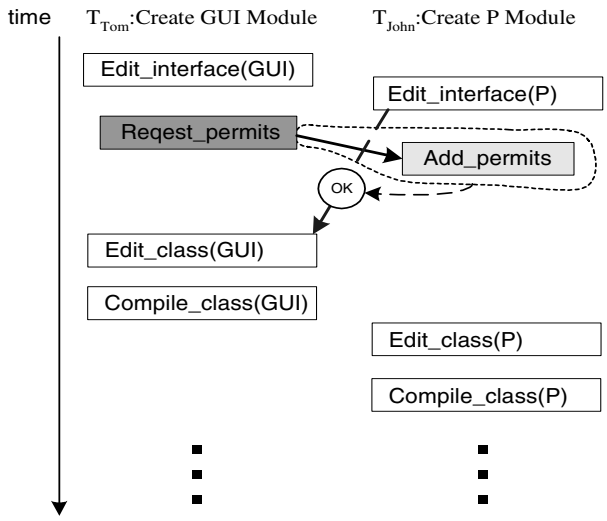


Figure 7.6 Correct scenario using conflicts and permits.

$commit(T_{\text{John}}) < commit(T_{\text{Tom}})$. And if T_{John} aborts then T_{Tom} must abort too – i.e., $AbortSet(T_{John})=\{T_{Tom}\}$.

Case 2: Permits and Demands only

This case only combines demands and permits. Permits identify allowable interleavings while demands specify steps that must appear. Since this assumes that conflict rules do not exist, permits are used to allow accesses to data that are controlled via locking. This implies that we will not be able to fully exploit the benefit of advanced operations, and so be unable to customise conflicts. The reason is that we would not be able to reason about conflicts appearing beyond read/write.

Nevertheless, we may regard locks as a specialisation of conflicts. Hence, this case is a special case of Case 4.

Case 3: Conflicts and Demands only

This case is equivalent to that of Nodine and Zodnik (1992). This combination may be applied to achieve correctness. However, since we may not directly refine a conflict definition (also true for the original conflicts from (Nodine and Zdonik 1992)), there is no way to relax any prespecified conflict rules if the requirements change. Nevertheless, it is often preferred to allow some limited number of transactions to disregard some specific conflicts rather than change these conflicts for all executing transactions collectively. Although cooperation could be useful for Tom and John, allowing a third party to observe the changes made may not necessarily be convenient, since the results may still be incomplete. In other words, removing some specific conflicts, thus opening up for all the involved transactions, may sometimes be inconvenient

with respect to consistency of the final results. Therefore, we may also need permits in addition to conflicts and demands.

To illustrate, consider the coding example involving Tom and John in Figure 7.7. Due to our conflict rules and the lack of permits, Tom and John may not execute *Edit_interface(P)* and *Edit_Class(GUI)* concurrently. This means that only one of the two engineers can update both *P_i* and *GUI_c* at a time. For this reason, Tom and John agree that Tom should do the necessary updates on *P_i* as well as *GUI*_c, while John modifies *P_c*.

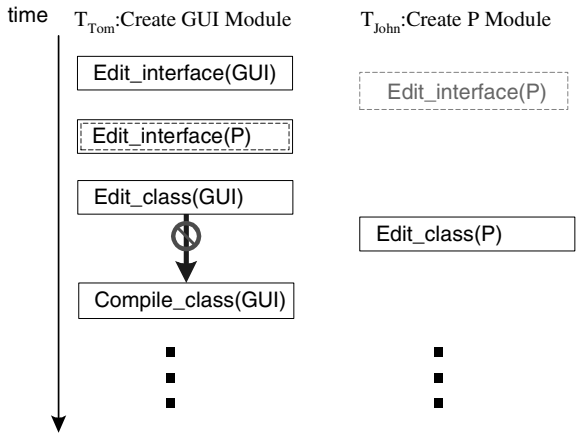


Figure 7.7 Invalid scenario according to conflicts and demands constraints.

This far, the interaction between Tom and John's transactions is legal. However, to make sure that all interface information included in *GUI_c* are complete, before producing the object code of *GUI*, Tom has to browse the object code of *P*. Therefore, the scenario in Figure 7.7 must be considered invalid. Tom has to wait until the object code of *P* is available.

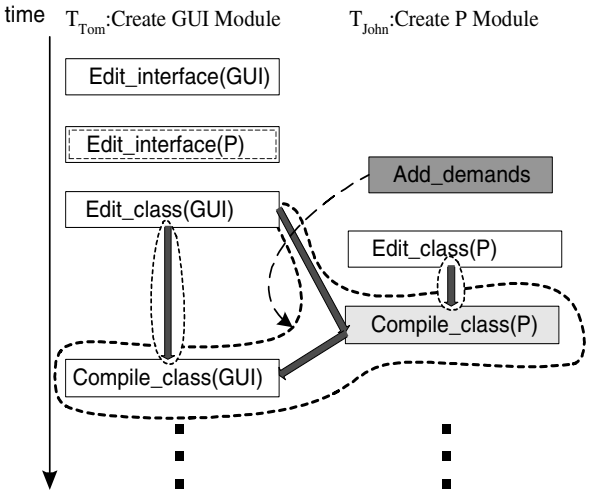


Figure 7.8 Valid scenario following conflicts and demands constraints.

Hence, Tom and John need a demand specifying that before *Compile_class(GUI)* can be executed – after *GUI* and *P* are modified, *Compile_class(P)* must be performed. As a result, the interactions depicted in Figure 7.8 represent a valid sequence of actions, using both conflict and demand constraints. A discussion of demand additions (the thick lines) is provided in Section 7.3.4.

Case 4: Conflicts, Permits and Demands

The main conclusion of the discussions in the three cases above is that we may need all the three constraints to both ensure correctness and enable some degrees of flexibility – i.e., controlled sharing of data. To illustrate this case, recall our coding example with Tom and John from Case 1, and consider the scenario in Figure 7.9.

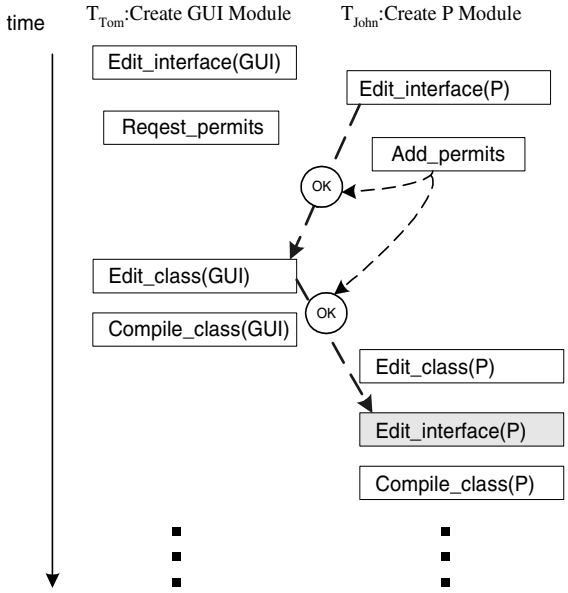


Figure 7.9 Illegal scenario due to incompleteness of conflicts and permits.

Suppose that John has to make some modifications to *P_i* after Tom has created the *GUI*-class. Because of the prevailing permits (see Case 1), this is okay. However, due to these modifications, the *P_i* incorporated by Tom is no longer up to date. This implies that Tom ought to re-execute *Edit_class(GUI)* to reflect the changes made to *P_i.* A natural way to ensure this is to add a demand enforcing the execution of *Edit_class(GUI)* after *Edit_interface(P)*.

Figure 7.10 shows a scenario following this rule. Here, the commit dependency between T_{Tom} and T_{John} still applies, meaning that after the first *Compile_class(GUI)*, Tom must wait for John's commit before he can terminate too.

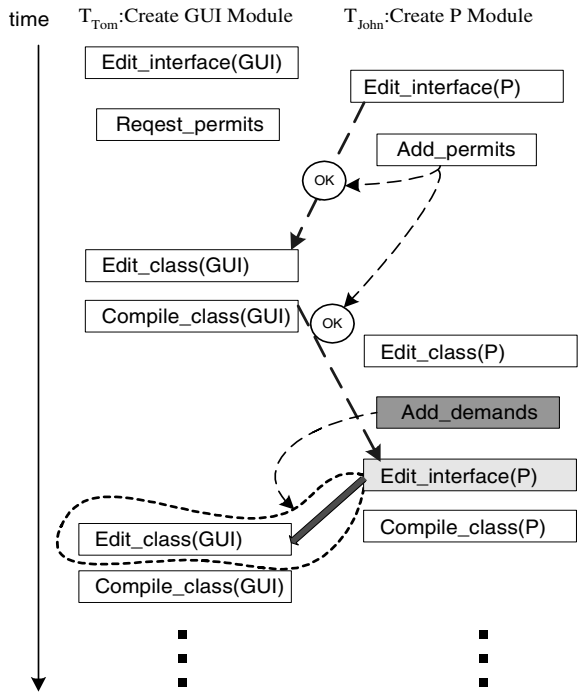


Figure 7.10 Legal scenario using conflicts, permits and demands.

7.3.4. Handling dynamic re-specifications of correctness constraints

We have several times stressed the ability to modify the three constraint types – i.e., conflicts, permits and demands – at runtime. Table 7.6 summarises the conclusions drawn from an analysis of the possible adjustments.

	Addition	**Removal**
Conflicts	N/A[a]	N/A[a]
Permits	Always okay	May cause difficulties
Demands	May cause difficulties	Always okay

Table 7.6 Properties of the execution constraints.

a. Conflicts are not modifiable.

7.3.4.1. Conflicts

Conflicts should not be modifiable during runtime to avoid unmanageable complexity. However, indirect refinements are still possible. For instance, user-controlled locks can be used (see Section 7.2.1) to gain more restriction, and permits can be added to gain more flexibility.

7.3.4.2. Permits

Addition of permits

Apart from the concurrency anomalies that may occur, and which are managed separately (see Section 6.4.1), the introduction of permits at runtime does not cause any problems since this will only affect operations that will be performed in the future.

To handle a new permit relationship – i.e., *permit*(t_j, t_i, *Op_set*, *Ob_set*), allowing t_i and t_j to perform a set of operations *Op_set* on a set of objects *Ob_set*, a specification manager (see Section 8.3.1) executes the following steps:

Add_permit(t_j, t_i, *Op_set*, *Ob_set*)

(1) Check whether conflicts for *Op_set* and *Ob_set* exist. If this is true, define the permits.

(2) If no conflicts exist, check whether the data in *Ob_set* are locked:

  (a) If the lock type is user-controlled, request degradation. If this can be granted, degrade the lock to a collaborative lock and define the permits.

  (b) If the lock type is collaborative, define the permits.

(3) Otherwise, no permits are necessary.

Removal of permits

In contrast, the removal of a permit relationship may not always be straightforward. Difficulties may arise due to existing conflict specifications. For example, some operations in the *Op_set* may be conflicting, thus causing an execution to be invalid. However, this only applies if the involved operations have already been executed. To manage the removal, the specification manager executes the following steps:

Rem_permit(t_j, t_i, *Op_set*, *Ob_set*)

For all operations in *Op_set*, check for conflicts. For each conflict found, check in the log whether the operation has already been executed:

  (a) If the operation is found in the "log", report the conflict to the user, providing options for further actions: (i) Invalidate (through abort), (ii) Ignore (not recommended), or
  (iii) Cancel (the removal).

  (b) If it has not been executed, remove the permits and return with success.

It is important to note that given the length of transactions and the amount of data that these transactions may touch, maintaining the information concerning permits may become a difficult and complex task. To cope with this complexity, the management of permits must be kept as simple as possible.

A proposed solution here is maintaining the permit information in a "database" of permits. Each permit "relationship" for two specific transactions is associated with table entries consisting of quadruple of the involved transactions, operations, and objects. Each operation has a flag telling whether it has been executed or not. When a permit is inserted to the "database", the execution flags for all involved operations are unset (false). Then, when a transaction executes a specific operation, the execution flag for that operation will be set (true).

So, when a permit is to be removed, the search in the "log" in (a) means issuing a query on the permit "database" and checking whether there are operations that have the execution flags set or not. When a transaction terminates, one or more (associated) permits will be removed from the "database".

7.3.4.3. Demands

Addition of demands

Although there are no problems adding permits, the introduction of demands is not always so straightforward. The following discussion considers different cases and situations that the specification manager must handle when a user adds demand constraints.

Case 1: Force an occurring operation Op_k in-between operations Op_i and Op_j

An *In-Between* (Op_i, Op_k, Op_j) constraint requires that if all three operations Op_i, Op_j and Op_k occur, then Op_k must succeed Op_i and precede Op_j in sequence – i.e., $\exists(Op_i, Op_j, Op_k), (Op_i < Op_k) \wedge (Op_k < Op_j)$. This is illustrated with a linking graph in Figure 7.11.

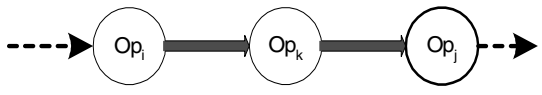


Figure 7.11 Illustration of an In-Between constraint.

See Case 3 in Section 7.3.3.2 for an application of this constraint type. The worst case here is when such an addition is requested just before Op_k is to be executed. It is okay if Op_j has not already been executed (irrespective of whether Op_i has been executed) – it is just a matter of adding the linking requirements and scheduling the involved operations appropriately (if all three

occur). But if Op_j has already been executed, it has to be re-executed – with possible recursive effects – when Op_k is executed (if Op_i also occurs).

Case 2: Force a non-occurring operation Op_k to occur and follow operation Op_i

An *Occur-Follow (Op_i, Op_k)* constraint requires that if a specific operation Op_i occurs, then the given operation Op_k must also occur and follow Op_i in sequence – i.e., $\exists Op_i \; \forall Op_k, (Op_i < Op_k) \wedge (Op_i \rightarrow Op_k)$. This is illustrated with a linking graph in Figure 7.12.

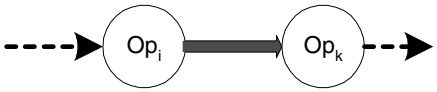


Figure 7.12 Illustration of an Occur-Follow constraint.

See Cases 3 and 4 in Section 7.3.3.2 for three applications of this constraint type. Such an addition typically involves forced scheduling of operations. The only recursive effects here would be other forced executions but no re-executions.

Note that operations may be involved in several demand constraints – of both types even – at the same time.

Removal of demands

Just like adding permits, the removal of a demand does not pose any difficulties. This is because occurring patterns of operations are not prohibited by nonexistent demands.

7.4. Integrating workspace management

Integration of the workspace concept in CAGISTrans was motivated by the need to facilitate the management of sharing in data intensive systems as well as to further widen the application areas of the CAGISTrans framework.

7.4.1. Flexible workspace support

Several extended operations were listed in Section 6.2.2 that become necessary as a result of the flexible workspace management needed in CAGISTrans. Recall that the operations provided are *write-check-out (wco)* and *read-check-out (rco)*, distinguishing the intentions behind the check-out operation, *upward-check-in (uci)*, checking in data from any level to the next level, *check-in (ci)*, checking in data from any workspace to the public workspace, *refresh*, updating a local copy of data with the one residing in the parent workspace, and data manipulation operations such as read, write, insert and delete, plus di-

verse advanced operations (see Section 7.3.2). Note that the main difference between *uci* and *ci* is that when a user performs *uci(obj),* he/she only puts a copy of *obj* from his/her workspace into a parent workspace. He/she still has the ownership on *obj.* But, with *ci(obj), obj* is moved to the public workspace, and the ownership to *obj* is released. This distinction is similar to that proposed in the literature (Kim et al. 1984, Bancilhon et al. 1985), allowing an object to be checked-in either to a semi-public workspace or a public workspace. However, as mentioned earlier, unlike the previous approaches, here the connection between the transaction structure and the workspace operations is not considered, allowing check-in to either public or group workspace to be performed independently of the current transaction structure. Moreover, the coordination of these operations is performed differently compared to the previous approaches. Here the workspace operations are coordinated through the constraint tools in CAGISTrans.

To illustrate the use of these operations, consider our coding example. Figure 7.13 incorporates in its right-hand side the workspace operations needed for the interaction between Tom and John corresponding to Figure 7.10. This figure is again an extension of Figure 7.3. The operations in shaded boxes are operations executed on the GUI module, while the other operations are executed on the P module.

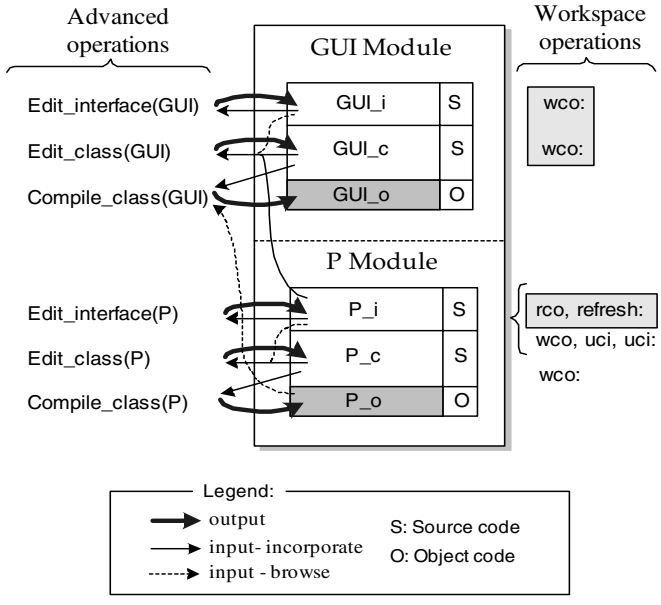


Figure 7.13 Illustration of workspace operations applied on GUI and P modules.

The detailed interaction between Tom and John, illustrating the use of the workspace operations, is depicted in Figure 7.14 and Figure 7.15.

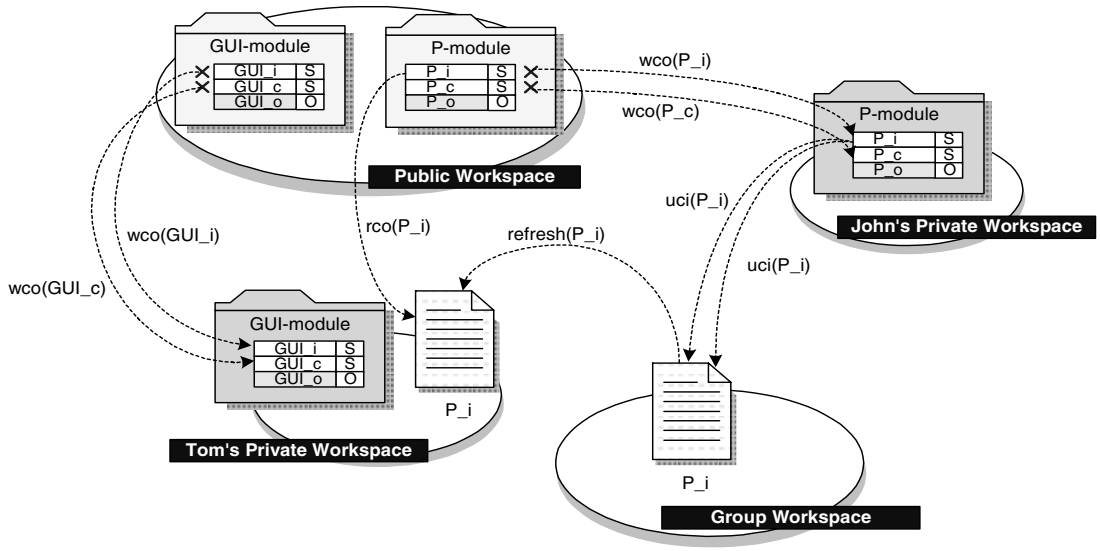


Figure 7.14 Illustration of workspace Tom and John's interactions.

Before editing the GUI interface (*GUI_i)*, Tom must check out *GUI_i* for write – i.e., *wco(GUI_i)*. This allows him to make all the necessary changes to *GUI_i*. John does the same with *P_i*. When John is finished, he updates the copy of *P_i* to his and Tom's group workspace by issuing *uci(P_i).* Here, John

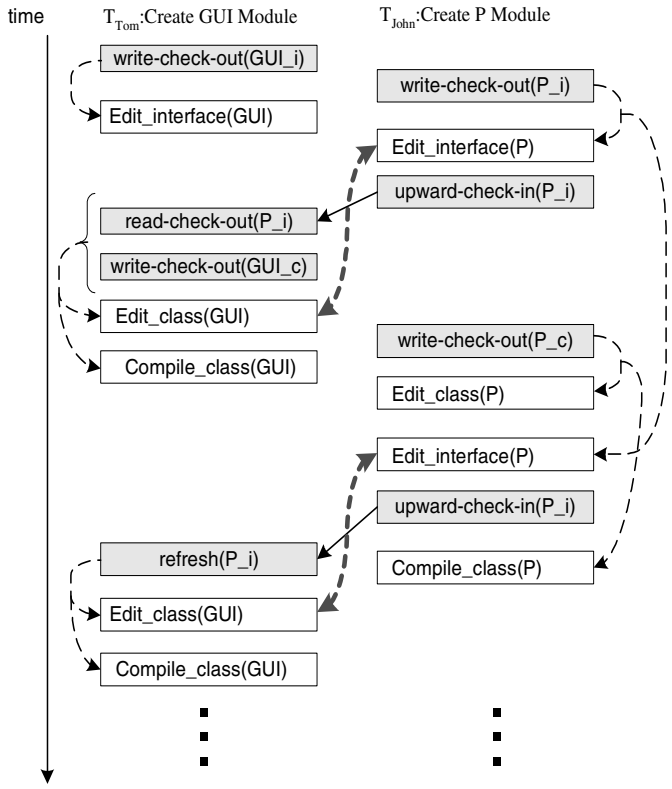


Figure 7.15 Illustration of the use of workspace operations.

chooses *uci* rather than *ci* since he knows that *P_i* is still incomplete. Therefore, making *P_i* publicly available is not reasonable yet. *P_i* is useful to Tom, though. Actually, Tom is now ready to edit the GUI class (*GUI_c*). For this, he needs a copy of *P_i* so he issues *rco(P_i).* Thereafter, he checks out *GUI_c* for write – i.e., *wco(GUI_c).* Now, while Tom is performing his updates on *GUI_c*, John finds out – after making some changes to *P_c* – that he must do some further changes to *P_i*, which he afterwards checks in to the group workspace once more. As Tom is forced to re-edit *GUI_c* (see Figure 7.10), he must update his copy of *P_i* with the latest changes. Since Tom already has a copy of P_i, he accomplishes this by issuing *refresh(P_i),* rather than another *rco(P_i)*. Hence, Tom will now update GUI_c to reflect the changes to P_i.

7.4.2. Correctness insurance and coordination

To ensure correctness and coordinate interactions through workspaces, we use user-defined constraint tools to specify prohibited interactions, permitted interleavings, and required sequences of operations, as described in Section 7.3.3.

To illustrate this, let us assume that the commutativity (conflict) rules from Section 7.3.2 apply and consider Figures 7.6, 7.10 and 7.15. Without the permit relationship from Figure 7.6, when Tom issues *rco(P_i)*, the execution manager would have to request him to wait until John has committed his updates on *P_i*. But, with the indicated permit relationship, upon request, John will be requested to make available his partial results by issuing *uci(P_i)*, allowing Tom to proceed with his task.

Further, John's repeated processing of *P_i* will affect Tom's processing of *GUI_c*. As indicated in Figure 7.10, this triggers a demand relationship too. This again will force the execution of Tom's *refresh* operation, after the new copy of *P_i* is made available through John's *uci* operation.

7.5. Chapter summary

This chapter has described the theory, the specification and the design of our CAGISTrans framework.

It has addressed the dynamic properties of cooperative work by providing transactional support that not only can be tailored to fit different needs, but can also be adapted to changes in the actual environments. The fundamental approach is to exploit the beneficial features from existing transaction models and frameworks, and extend these to form our CAGISTrans framework. An explicit comparison to related work is provided in Section 11.3.

Chapter 8
A System Supporting the CAGISTrans Framework

8.1. Introduction

Cooperative work environments are heterogeneous in addition to being dynamic. Therefore, openness is crucial. Previously, we indicated the necessity of supporting changes in the environment, while users are executing their transactions. Along the same line, the necessity of providing support for a wide range of resource management systems was stressed. Motivated by this, the CAGISTrans transaction management system has been developed as middleware (see Section 6.5). The objective here is to provide a system built on top of available DBMSs, which at the same time can offer features that extend these DBMSs in terms of distribution support, increased resource availability and database independence, as well as providing flexible cooperative transaction management support.

This chapter presents of the design of a CAGISTrans system architecture. Thus, it particularly shows how the CAGISTrans framework deals with problems incurred by the heterogeneous nature of cooperative environments, in addition to those incurred by their dynamic nature. To this end, this chapter is organised as follows. Section 8.2 outlines system requirements based on the discussion in Chapters 5, 6, and 7. Using this as a starting point, Section 8.3 outlines the CAGISTrans system architecture. A discussion of our experience from experimenting with patterns and conflicts is provided then in Section 8.4. This states the main reasons for the selection of our correctness constraint tools. Section 8.5 then discusses the relation of our CAGISTrans system to other sys-

tems. Finally, Section 8.6 summarises how our system satisfies the requirements set up in Section 8.2.

8.2. System requirements

Our system requirements are derived from a combination of the cooperative work requirements in Section 5.3 with the CAGISTrans framework objectives derived from the results in Chapter 7. The following is the list of requirements a CAGISTrans system must meet in order to adequately support our CAGISTrans framework. See also Table 8.1 for a summary.

SR1 *Support for transaction model specification.* A CAGISTrans system should provide an environment facilitating specification of transaction models.

- **SR1.1** *Support for characteristics specification*. A CAGISTrans system should allow a user to specify desired "transaction properties" – i.e., configuration of atomicity and isolation properties, and enable desired correctness constraints – i.e., enabling of system dependent or user-defined criteria.
- **SR1.2** *Support for transactional behaviour specification*. A CAGISTrans system should allow specification of transactional behaviour. The specification environment should allow users to define advanced operations, linking these to basic commands provided by the underlying environments.
- **SR1.3** *Support for definition of correctness constraints.* If user-defined constraints are enabled (see SR1.1), a user should be able to specify conflicts, initial permits and initial demands. Finally, a CAGISTrans system should make it possible for a user to modify any specified permits and demands at runtime.

SR2 *Support for transaction execution at runtime*. A CAGISTrans system should provide a platform for execution of transactions according to the specified transaction model.

SR3 *Support for transaction execution awareness*. A CAGISTrans execution environment should provide notification mechanisms that at least indicate the start of a transaction, termination of a transaction, acquisition of locks and object changes.

Req. ID	Requirement description
SR1	A CAGISTrans system should provide support for specifying transactions models.
SR1.1	A CAGISTrans system should support the specification of transaction characteristics.
SR1.2	A CAGISTrans system should support the specification of transactional behaviour.
SR1.3	A CAGISTrans system should support the definition of correctness constraints whenever this is enabled.
SR2	A CAGISTrans system should provide a runtime environment for execution of transactions applying a specific model.
SR3	A CAGISTrans system should provide awareness associated with execution of transactions.
SR4	A CAGISTrans system should support integrated workspace management.
SR4.1	A CAGISTrans system should allow mapping of workspaces to repositories.
SR4.2	A CAGISTrans system should provide extended workspace operations.
SR4.3	A CAGISTrans system should provide workspace information services.
SR5	A CAGISTrans system should support at least three types of resource management systems.

Table 8.1 Summary of CAGISTrans high level requirements.

SR4 *Provision of workspace support*. A CAGISTrans system should enable integrated workspace management, fulfilling the following sub-requirements:

SR4.1 *Mapping of workspaces to repositories*. A workspace should be mapped to repositories – including DBMSs, file systems and Web servers – in order to facilitate data management.

SR4.2 *Provision of workspace operations.* CAGISTrans extended workspace operations – including, at least, write-check-out, read-check-out, check-in, upward-check-in and refresh – should be provided by the CAGISTrans execution environment.

SR4.3 *Provision of workspace information*. A CAGISTrans system must provide the user with information concerning available workspaces, and the types, owners and contents of such workspaces.

SR5 *Openness with respect to resource base support*. A CAGISTrans system should provide interfaces allowing the use of at least three resource storage types – i.e., DBMSs, file systems and Web servers.

Fulfilling these requirements is a key to the design and implementation of our CAGISTrans architecture.

8.3. The CAGISTrans architecture

This section discusses the architecture of a CAGISTrans system. Figure 8.1 depicts this architecture. A natural implication of the distinction in Figure 6.2 is to provide an environment that allows both design time specification of transaction models and runtime management of transaction execution.

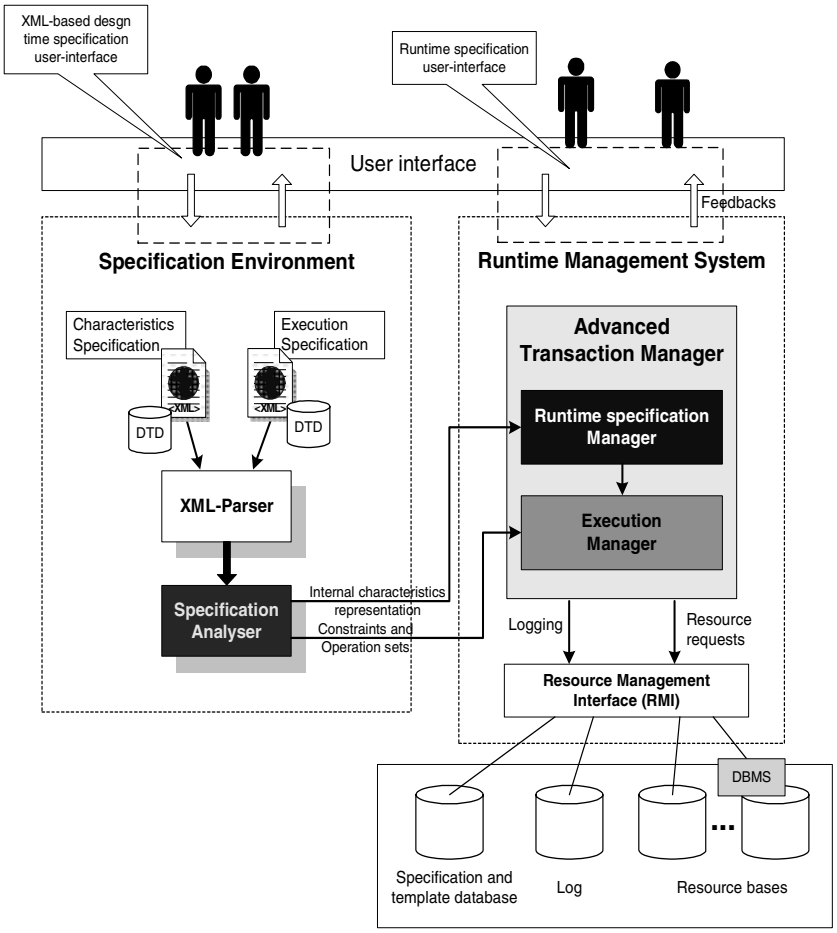


Figure 8.1 Architecture of the transaction management system.

Therefore, the transaction management system is divided into two separate environments, consisting of a *specification environment* and a *runtime management system*. In order to meet SR1 and SR2 also necessitate such a separation.

The following sections explain these components.

8.3.1. The specification environment

An important aspect of our system is that all specifications are encoded in XML. Therefore, the most dominant components of this environment are those used to define, parse and validate transaction model specifications.

In this environment, a model designer specifies the desired transaction characteristics and the initial execution constraints and operation sets, before he/she executes his/her transactions. These are, thereafter, passed through an XML parser, which checks the specification against prespecified DTDs – document type definitions. Next, a specification analyser walks through the specification and translates the generated elements from the XML parser into internal representations of transaction characteristics – i.e., dependency formation rules, enabled correctness criteria and applied policies, and operations sets and execution constraints – i.e., conflict table entries, initial permit sets, and initial demand rules (see Chapter 7).

Referring to Figure 8.1, we have implemented an *XML*-based design time specification of transaction characteristics based on definitions of the desired "non-ACID" requirements. In addition, a complete *XML/DTD*-based language has been implemented. Here, DTD plays the role of a language dictionary. To parse specifications, we have integrated an *XML-parser* from IBM (the XML4J-parser[1]). Analysing and translating the outputs from the XML parser – i.e., the *specification analyser (SA)* – is also an integrated part of the CAGISTrans prototype. An overview of the implementation status of the components is provided in Table 8.2.

8.3.2. The runtime management system

The representations from the specification environment are used by the runtime management system to control and manage the execution of transactions. Its main component is the *advanced transaction manager*, responsible for making sure that the execution of transactions conforms with the transaction characteristics and execution specifications from the specification environment. This manager again consists of two sub-components; the *runtime specification*

1. See http://www.alphaworks.ibm.com/aw.nsf/techmain/xml4j.

Blocks		Implemented	Comments on not fully implemented components
User interface	Design time specification	60%	Fully functioned graphical user interface is not finished yet. Using standard text editor to hard-code the specification.
	Runtime specification	100%	
Characteristics and Execution – XML & DTD		100%	
XML-Parser		100%	
Specification Analyser		60%	Automatic check for supported model is not available.
Runtime Specification Manager		80%	Handling of advanced conflict rules is only designed.
Execution Manager		75%	Realisation of dynamic restructuring is only designed. Enforcing advanced conflict rules is not supported.
Resource Management Interface		90%	Interfaces to standard ACID transaction using JTA/JTS are implemented, but not yet integrated and tested.

Table 8.2 Overview of implemented components.

manager (RSM) and the *execution manager (EM)*. Before transactions are executed the RSM checks that the specifications can be supported. The CAGISTrans system is designed to operate on top of DBMSs. Thus, if a user chooses to rely on an underlying DBMS to handle correctness control, the RSM checks – through the *resource management interface (RMI)* – that such support is present. Unfortunately, many existing DBMSs have their own ways of implementing concurrency control. For example, it is often impossible to access the lock tables that are managed by these systems. Instead, some systems may allow locks to be explicitly acquired and released through SQL-statements[1]. Therefore, "co-operative" management of concurrency control between a CAGISTrans system and DBMSs can only be achieved to a restricted degree. A way to overcome this limitation is allowing the underlying DBMSs to do the required control of sharing. As such, correctness of data managed by the DBMSs can be assured in each system; but achieving global consistency (across the systems) is still an

1. See for example the GET_LOCK and RELEASE_LOCK statements in MySQL.

important challenge that must be further considered. This issue has been a subject for intensive research in the last couple of decades. Several solutions have been proposed the literature (Breitbart et al. 1995, Mehrotra et al. 1998, Mehrotra et al. 2001). However, it is widely agreed that managing consistency across multiple heterogeneous and autonomous DBMSs is an issue that still deserves further attention.

From transaction commitment point of view, CAGISTrans can implicitly affect the way underlying DBMSs handle commitment by enabling or disabling auto-commit. In fact, there are database systems that allow this specification through JDBC drivers.

From these perspectives, checking the underlying DBMSs for the support provided means checking (1) which type of isolation level is supported for each DBMS and (2) whether auto-commit can be enabled or disabled as needed.

Alternative to relying on DBMSs, the user may want to rely on the advanced support provided by the CAGISTrans system. In that case, the RSM will manage the execution of user transactions in cooperation with the EM.

Further, we recall that new specifications may be introduced during runtime. The RSM provides the necessary support for the required modifications. For example, when new constraints are to be introduced, the RSM first checks the actual specification. Then it gives the user all the necessary information on the actions that must be taken. To illustrate this consider the addition of new constraints to the current set of demands discussed in Chapter 7. The RSM checks the log – i.e., an *execution descriptor* – for all executed operations. If any new constraint may cause invalidation of some operations because a specific operation is to appear between two already performed operations, the RSM will inform the user about this, requesting him/her to choose the way to proceed – either to allow the invalidation or to cancel this introduction.

Validation of transaction executions is managed by EM. The EM uses the specified correctness criterion to control the execution of a specific transaction. The EM can be seen as a transaction scheduler in the sense that it is responsible for executing the transactions on the basis of requests. In addition, it is responsible for managing the execution of relevant management operations. This includes applying the prevailing dependencies, which are relevant when issuing *begin*, *abort* or *commit* (see Section 7.3.1). Further, the EM allows transactions having a nested structure to *spawn* new transactions upon request. And, if necessary, the EM also accomplishes *delegation* of operations among transactions.

As shown in Table 8.2, we have implemented a major part of the *runtime management system*. More specifically, we have designed and implemented the

user interface which supports interactive transaction initiation and termination. The RSM component is implemented to handle read/write-based conflicts only. Handling more advanced conflict rules has been designed, but this is still not integrated in the CAGISTrans prototype. Permits and demands are, on the other hand, part of the prototype implementation. The current implementation of RSM also supports specification of the two latter constraints during runtime. Further, the EM component allows validation of transaction executions using permits and demands. However, even though dynamic restructuring is designed, it is still missing from the EM prototype implementation. Finally, some advanced operations are integrated and supported by the EM. These include the operations used to handle access to workspaces that we elaborated in Section 7.4.

8.3.3. Ensuring correctness

In CAGISTrans, there are two possible ways to achieve desirable final results: enforcing serialisable execution and adopting user-defined correctness criteria.

The serialisability criterion is mainly supported by underlaying resource management systems. Therefore, the CAGISTrans system provides this criterion through the interfaces to serialisable resource management systems, as shown in Figure 8.2. These interfaces are realised with JDBC – Java Database Connectivity[1] – and the JTA/JTS – Java Transaction API[2] / Java Transaction Service[3] – interfaces. Using the JDBC interface, we are able to specify the isolation level of the underlying DBMS connected to the CAGISTrans system. This

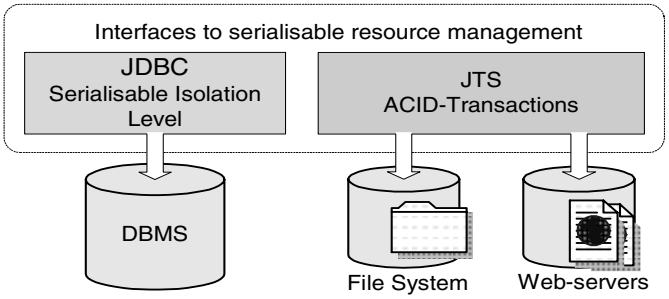


Figure 8.2 Interfaces to serialisable resource management systems.

1. See http://java.sun.com/products/jdbc/.
2. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system. See http://java.sun.com/products/jta.
3. JTS specifies the implementation of a Transaction Manager which supports the JTA, based on the CORBA OTS (Object Management Group 1998). See http://java.sun.com/products/jts/.

means that we can enforce serialisable execution on that DBMS by choosing the serialisable isolation level – i.e., "TRANSACTION_SERIALIZABLE"[1]. On the other hand, if we use non-DBMS resource providers – such as standard file systems and Web servers – the CAGISTrans system will use the JTA/JTS interface to enforce serialisable execution. Note, however, that this enforces a full ACID execution, making it impossible to use relaxed atomicity too. Note that JTA/JTS is specified to support the 2-phase-commit (2PC) protocol[2], which allows *atomic* commitment of transactions executed on distributed resource bases.

Unlike serialisability, management and enforcement of user-defined criteria is achieved through the CAGISTrans system itself. Figure 8.3 shows how we implement this in CAGISTrans. As depicted, we are still able to use a database as a resource provider through a JDBC interface. But, now the isolation level is "TRANSACTION_NONE", indicating that all transactions are handled by the CAGISTrans system. JTA/JTS, on the other hand, only allows ACID transactions. Therefore, a JTA/JTS interface for extended transactions is not applicable. Rather, the CAGISTrans system takes over the responsibility for handling advanced transactions through the specification analyser (SA) and the execution manager.

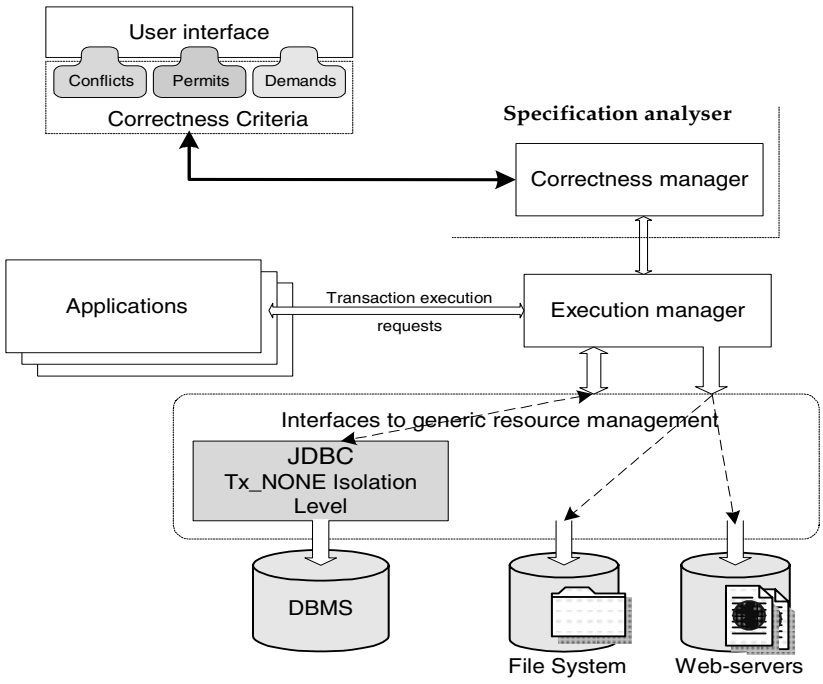


Figure 8.3 Managing user-defined correctness criteria.

1. See http://java.sun.com/products/jdbc/.
2. The principle of the 2PC protocol is discussed thoroughly in the literature (Bernstein et al. 1987, Özsu and Valduriez 1991).

The user specifies the desired correctness criterion through a user interface, allowing the specification of conflicts, permits, and demands. These are, thereafter, handled by the *specification analyser* (SA) which checks their validity and consistency (see Section 7.3.1.2 for a discussion). The SA cooperates with the EM to ensure that the transactions meet the specified constraints when they are executed.

Ideally, users should not be allowed to access the underlying resource systems unless they do this through the CAGISTrans interface. Failure to do so might jeopardise the overall consistency preservation on objects being shared. However, the challenge is to provide an access control mechanism that is efficient enough to fully restrict all accesses to the systems to those that only go through CAGISTrans. A solution to this would depend very much on whether the underlying systems allow data to be made accessible only to the CAGISTrans resource interface.

Referring back to our implementation overview in Table 8.2, the JDBC interface to DBMSs is supported by the current CAGISTrans prototype. For example, we use a MySQL[1] server as database for storing management and control information. This database server was chosen due to its simplicity in use and implementation. Further, we have tested the resource management interfaces against a PostgresSQL[2] DBMS. We are also currently investigating use of the IBM DB2[3].

8.3.4. Mapping between workspaces and resource bases

Our workspace concept can be regarded as an abstract concept. This means that physical workspaces do not exist by themselves. Instead, they are logically mapped to underlying resource bases, as illustrated in Figure 8.4.

As depicted in this figure, objects reside on several types of repositories, including databases, file systems and Web servers. Then, when an object is checked-out or checked-in, it is "marked" with where it will logically belong – i.e., to a *private workspace*, *group workspace* or *public workspace*. Thus, to facilitate this mapping, we define an object as a tuple `(id, type, ws-state, address, owner)`, where

- `id` is a unique object identification,
- `type` is the object kind – i.e., `file`, `web-doc`, or `relation` (see Figure 8.4),

1. See http://www.mysql.com/.
2. See http://postgresql.readysetnet.com/.
3. See http://www.ibm.com/db2/.

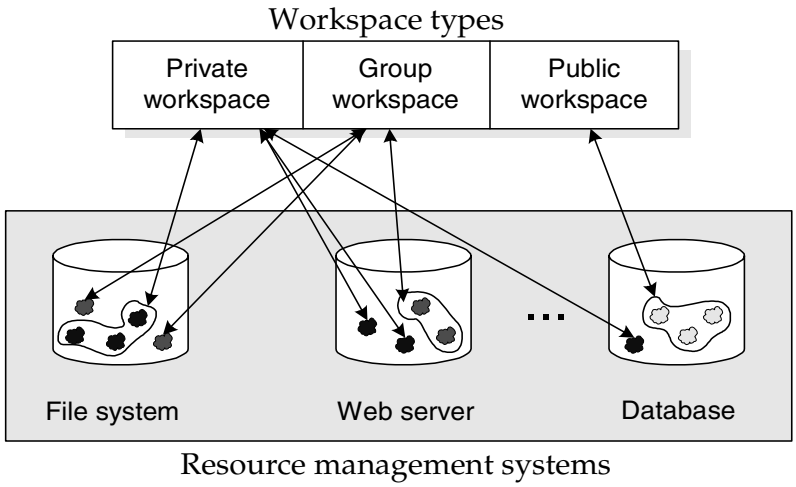


Figure 8.4 Illustration of mapping between several resource management systems and workspaces.

- `ws-state` identifies the type of workspace that the object is checked-out or checked-in to – i.e., `ws-state` may be `private`, `group` or `public`,
- `address` denotes the physical location of the object:
  - If the object type is file, the address will be a file path – i.e., "`file://<path-name>`"
  - If the object type is web-doc, the address is either an IP-address or an HTTP URL – i.e., "`http://<ip-address>`" or "`http://<web-url>`"
  - If the object type is relation, the address is a database URL – e.g., determining the JDBC driver; i.e., "`jdbc://<database-protocol>:<database-host>/<database-name>`",
- `owner` identifies the current owner of the object – i.e, *username* or *groupname*. If `ws-state` is `public`, the owner is empty.

Since workspaces are only logically defined, they can be created as needed depending on the current cooperation situation. They are simply determined by the objects' `ws-state` and `owner`. This means that objects residing in the private workspace owned by Tom are all objects such that `ws-state='private'` and `owner='tom'`. Objects in a group workspace `Gi` are all objects such that `ws-state='group'` and `owner='Gi'`. Objects in a public workspace are all objects such that `ws-state='public'` and `owner=' '`.

The above mapping facilitates the use of our extended workspace operations. This means that when an object is checked-out from or checked-in to a

workspace, it suffices to update the objects `ws-state` and `owner` accordingly. The address will thus specify the link to the resource base to be used.

8.4. Experiments with conflicts and patterns

The ideas underlying the correctness constraints tools of our CAGISTrans framework have originally been inspired by conflicts and patterns from the cooperative transaction hierarchy model (Nodine and Zdonik 1992).

Before implementing our constraint tools, we first tried to implement *conflicts* and *patterns*. We did some experiments with specification of grammars for conflicts and patterns. From this we learned that meeting the dynamic requirements of cooperative work was not a trivial matter. First, we had to define a complete LR(0)-grammar for the patterns and conflicts, and generate a lexical analyser using – e.g., *lex* (Levine et al. 1992), and a parser using – e.g., *yacc* (Levine et al. 1992), before we could execute the involved actions. For this to be possible, however, we also needed complete transaction (operation) sets to be carried out. Moreover, change incorporation at runtime would not be possible as we first had to modify the grammar, then generate a new lexical analyser and a new parser, before execution of the actual transactions could start again. These are the main reasons for deciding to develop our own demands and a revision of conflicts.

An additional lesson learned from these experiments was that the complexity of the grammar increased proportionally with the number of transactions and operations involved. Such a complexity would make the cost of managing the transactions unduly high. This further supports our argument for allowing runtime adjustments. In fact, by making such adjustments possible, we might allow stepwise specification, thus coping, to some extent, with the aforementioned complexity.

8.5. Comparison with other work

The CAGISTrans system architecture distinguishes between a *specification environment*, providing a means for specification and validation of transaction models, and a *runtime management system*, offering support for execution of transactions and management of their behaviour during runtime. The main advantage of such a separation is the ability to reason about properties and behaviour of transactions before they are executed, allowing a model designer to customise his/her model as desired. Moreover, the ability to refine the model at runtime is useful as we may need to support new requirements – e.g., due to the evolutionary behaviour of the actual activity. There are other approaches

that apply similar separation of specification and runtime environments. Most comparable with our work from this perspective are TSME (Georgakopoulos et al. 1996) and TransCoop (de By et al. 1998). Compared to these, CAGISTrans has attempted to improve the dynamic support. In TSME a specification is tested with respect to whether it may be supported before it is applied. Once the specification passes, and the transactions are executed, there is no way to change the provided specification before the transactions are terminated or interrupted. Moreover, support for dynamic restructuring was beyond the scope of TSME. TransCoop realises the aforementioned separation by defining activity scenarios (de By et al. 1998) prior to transaction executions. Like TSME, once the scenario is defined – i.e., specified, validated, and compiled – the runtime environment does not provide any possibility for "on-line" refinement of the scenario definition.

Further, while both TSME and TransCoop were implemented as complete systems, our framework is built on the middleware principle, making it possible to provide support for a wide range of resource management systems.

Our system is implemented making use of both standard mature technologies such as XML[1] and Java[2] and advanced evolving technology such as Software Agents. We refer to Chapter 10 for a more thorough description and discussion of our general experiences with and specific use of these technological tools.

8.6. Meeting the system requirements

Requirement SR1 is met as follows. Our CAGISTrans system allows users to define the transaction properties and enables them to designate appropriate correctness constraints. For this reason, SR1.1 is satisfied. Further, SR1.2 is met by allowing users to specify the start and end of transactions interactively and allowing them to specify advanced operations, primarily those used in connection with workspace management. Conflicts based on read and write conflicts are currently supported, in addition to an extension with respect to compatibility rules based on workspace operations. Finally, the transaction system allows users to define initial permits and demands, and refine these at runtime, thus meeting SR1.3.

A CAGISTrans system has a runtime support system making it possible to execute transactions in accordance with specified transaction characteristics.

1. See http://www.w3.org/XML.
2. Java is a trade mark of Sun Microsystems. See http://java.sun.com.

This means that the transaction system allows scheduling of transactions following specific correctness constraints. Hence, SR2 is satisfied.

All transactions executed by a CAGISTrans system are given notifications informing them about the start or termination of a transaction. In addition, if a user specifies some relaxed isolation, transactions are given notifications specifying lock acquisitions and object changes. Through this, a CAGISTrans system satisfies SR3.

Requirement SR4 is satisfied as follows. Workspaces are mapped logically to the underlying repositories, hence simplifying data managements. Through this, SR4.1 is met. Provision of workspace operations – i.e., SR4.2 – is supported via workspace operations being implemented as default advanced operations. Information concerning existing workspaces, owner of these workspaces and their contents is available to users. Hence, SR4.3 is also supported.

SR5 requires a CAGISTrans system to be open, supporting at least three types of resource bases. A CAGISTrans system provides interfaces allowing the execution of transactions on top of DBMSs, file systems and Web servers. This means that our CAGISTrans system meets SR5.

Part III

IMPLEMENTATION AND ASSESSMENT

URN:NBN:no-2103

URN:NBN:no-2103

Chapter 9
Realisation of a CAGISTrans System

Proof of concept prototypes have been implemented to realise the CAGISTrans framework based on the architecture described in the previous chapter. These prototypes support major parts of this framework and architecture. However, since the purpose of the implementation was primarily to illustrate ideas, they might still lack robustness, user-friendliness, and performance.

This chapter is organised as follows. First, Section 9.1 presents the applied implementation infrastructure. Section 9.2 presents the prototype architecture built on this infrastructure. Finally, Section 9.3 discusses the possibility of using a CAGISTrans system in the context of agent-based groupware. It particularly focuses on the challenges that must be faced.

9.1. Implementation infrastructure

This section outlines the tools used in the implementation of the prototype. The prototype was mainly built using agent technology. The main reason for this was to facilitate the integration of distribution support – e.g., Web support. Moreover, the choice of using agents as a platform for developing a CAGISTrans system has been motivated by other agent-based components in our CAGIS context (see Chapter 10). Other relevant aspects, such as mobility and network latency, are discussed in Section 11.1.3.

9.1.1. Agent platform – IBM Aglets

At the time we started our first implementation, we decided to use the

IBM Aglets Software Development Kit[1] – ASDK for short – for designing mobile agent systems in Java[2]. One of the main features of aglets (denoting ASDK agents) that we found useful was their mobility – i.e., their ability to move in a network from one host to another. In addition, an aglet is able to communicate and interact with other aglets, thus being in some sense cooperative.

Another positive aspect of the ASDK is its simplicity. Actually, we found this development kit the simplest mobile agent framework among those available for development purposes. Some other frameworks were those discussed in Section 3.3. The ASDK package was easy to install. And more importantly, it was a straightforward mobile agent technology that fitted well into the Java programming language, which is the language used in our prototype.

9.1.2. Agent communication language – KQML

An important aspect of agents is their ability to interact with other agents. To make our prototype generic, and to allow interaction with external agents other than aglets, we decided to use the more or less de facto standard agent communication language (ACL for short) KQML – knowledge and query manipulation language (Finin et al. 1994). KQML is both a language and a protocol for exchanging information and knowledge among agents. This means that KQML defines both the format of the messages exchanged, and the protocol for how the messages are handled during runtime. A detailed overview and discussion of the KQML language is given in (Finin et al. 1994).

9.1.3. Realising KQML with JKQML

A major limitation of many existing mobile agent systems is their lack of support for a standard communication language. Most agent systems provide their own solutions for communications. Therefore, there is no way to make agents from one specific system talk with agents from another system. This has stressed the necessity of a widely accepted (standard) ACL. However, it seems that a commonly agreed upon standard is still missing. We have therefore chosen to rely on JKQML.

JKQML – standing for Java KQML – is a KQML application program interface (API) for Java developed by IBM Yamato Lab[3]. It provides a building block for constructing KQML-speaking agents. One of the main reasons JKQML is used in our work is its direct compatibility with the ASDK. With the lack of a standard platform, this approach has been the least complex with re-

1.See http://www.trl.ibm.com/aglets or http//www.aglets.org.
2.Java is a trademark of Sun Microsystem.
3.http://www.alphaworks.ibm.com/formula/jkqml.

spect to the adoption of KQML.

9.2. Prototype architecture

We have implemented a system to handle both the specification of transaction models and the execution of transactions. Figure 9.1 depicts the implemented system architecture for the CAGISTrans transaction management system.

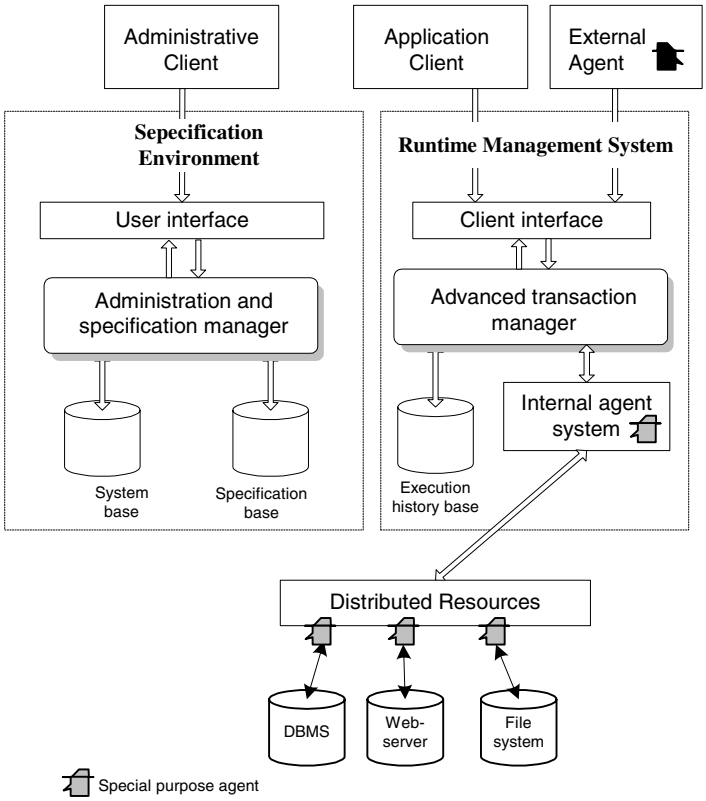


Figure 9.1 Implemented system architecture.

Conforming to the design of the CAGISTrans architecture (see Section 8.3), the main components of the prototype are divided into two separate environments consisting of *a specification environment* and *a runtime management system.* In addition, the system provides user interfaces and supports interfaces to back-end resource servers. Users may also access the CAGISTrans system through the external agents interface.

Now, let us walk through the architecture – including the user interfaces. There are three main types of interfaces in the CAGISTrans system. The first in-

terface is the *administrative client interface,* used mainly to provide the system administrative information. It consists of forms allowing a system administrator to do user setup – i.e., registration of new users including passwords and user names, and registration and setup of workspaces – i.e., creation of new workspaces containing owner information, types (private, group, or public), and physical addresses. The second interface is the *application client interface – i.e., CoopInterface.* This is the interface provided to cooperating users, allowing them to choose and connect to an available private workspace, group workspace and/or public workspace. The *CoopInterface* also provides the possibility for users to interactively initiate and terminate transactions, and to compose actions (operations) belonging to a specific transaction. Further, to allow awareness, this interface also provides a status window containing information on ongoing activities, participants in these activities and all relevant events.

The third interface is an interface to *external agents*. In the current CAGISTrans prototype, we have implemented this interface as part of the *CoopInterface*. The main reason for this is to increase the distribution support of a CAGISTrans system by making it possible for other non-CAGISTrans specific agents to have access to such a system. It allows users to access the CAGISTrans transactional services through aglets or other KQML agents. Section 9.2.3 provides an elaboration on this interface.

9.2.1. The specification environment

The purpose of the *specification environment* is to manage user setup and transaction model specifications. Its main component is the administration and specification manager, consisting of an *administrator manager (AM)* and a *specification manager (SM),* as illustrated in Figure 9.2. The AM is responsible for maintaining information from the administration client interface. For security and privacy reasons, all access to the system and its resources must be authenticated. Thus, the AM also includes the provision of an authentication service allowing users to log on to the system and connect to available workspaces. Note that only users participating in a specific activity may access workspaces for that activity. In the current version of the prototype all user and workspace information is stored on an SQL database sever. This database contains current

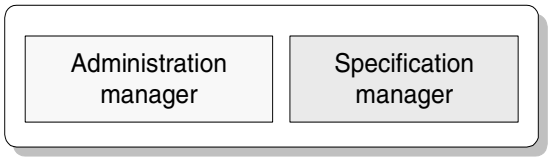


Figure 9.2 Components of the administration and specification manager.

users of the system and currently available and active workspaces. The SM, on the other hand, allows a model designer to specify the transaction characteristics and an initial specification of relevant sets of advanced operations, plus constraints to be applied – i.e., conflicts, permits, and demands, all being parsed and stored in XML format (see Section 8.3). SM is thus composed of an XML-parser – i.e., IBM's XML4J parser – and a specification analyser that are responsible for parsing and validating a specification document, and translating these into commands and rules that are passed to the advanced transaction manager for execution and control purposes. In the current version of our prototype, a model specification is produced using a general purpose editor with XML capabilities, instead of an integrated specification user interface. There the SM gets the specification XML-documents from disk rather than from a user interface.

9.2.2. The runtime management system

The *runtime management system (RMS)* consists of components that manage transactions at runtime. In addition, the RMS is responsible for making sure that transactions are executed according to the prevailing transaction model and all applicable constraints. The dominant component of the RMS is the *advanced transaction manager (ATM).* In accordance with our CAGISTrans architecture in Figure 8.1 in Section 8.3, RMS consists of two main components (see Figure 9.3): the *runtime specification manager (RSM)* and the *execution manager (EM).*

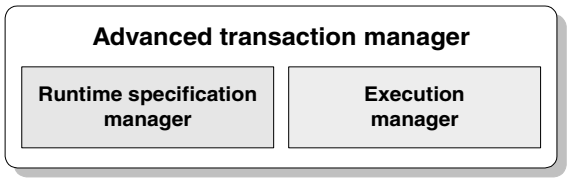


Figure 9.3 Components of the advanced transaction manager.

As the CAGISTrans system is aimed at supporting runtime customisation of transaction models, the main function of the RSM is to manage specifications at runtime, including those that are introduced while transactions are being executed. The EM works like a transaction scheduler. It manages the execution of transactions according to user requests. To make sure that the results from transaction executions are correct, the EM cooperates with the RSM to validate executions based on read/write conflict rules, existing permit relationships and prevailing demands. This also conforms to the specified CAGISTrans architecture. Further, the EM maintains a status database used for monitoring the

execution of transactions. This database is also used by the RSM to store the currently used transaction models.

9.2.3. Interface to external agents

The main purpose of the external agent interface is to allow external agents to access resources administrated through the CAGISTrans system. Implicitly, this means that if several agents cooperate via the same objects, our system provides transactional services which support the cooperation.

Currently, we have implemented a system that allows both aglets and other KQML-speaking agents to access such CAGISTrans services. This allows agents assisting users in a cooperative process to follow a cooperative policy specified for that cooperation. This means that when one or more agents access a back-end resource server they will follow the prevailing correctness criteria specified for their interaction.

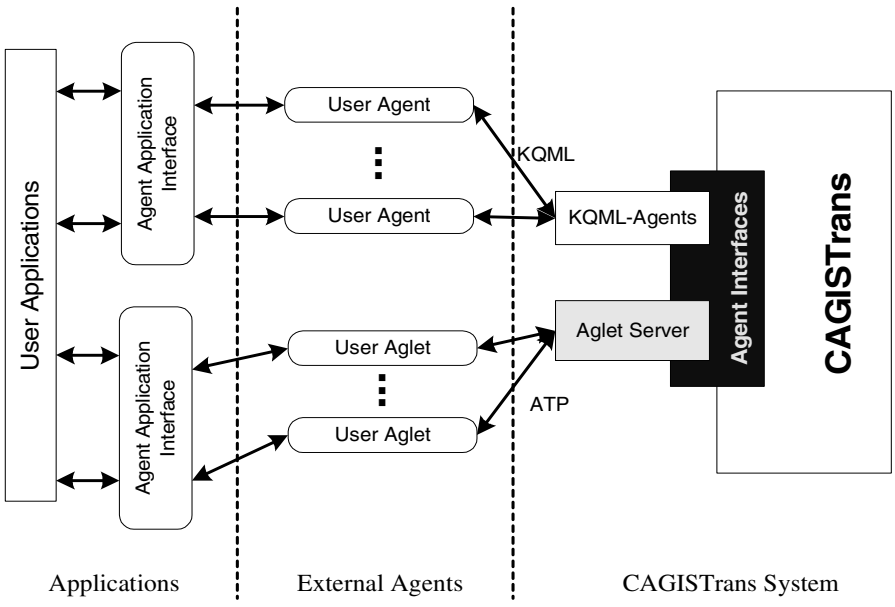


Figure 9.4 Interface to external agents.

Figure 9.4 depicts how the external agents interface in CAGISTrans is realised. CAGISTrans currently supports two categories of agents: KQML-speaking agents and aglets. KQML agents communicate with the CAGISTrans system through a facilitator (see Figure 9.5), using the KQML communication protocol. A facilitator is a service provider maintaining information about active system agents – i.e., special purpose agents forming part of the CAGIS-

Trans system itself – that accomplish agent requests. A system agent, which is an aglet, advertises all available services to the facilitator along with its physical address and symbolic name. Such services include executing operations through CAGISTrans, mediating transparent accesses to underlying distributed resource bases, and presenting notifications for awareness purposes. This means that a client application can use agents to request executing operations such as workspace operations through the CAGISTrans agent interface. They can also be used to access remote resource servers where documents or objects used in cooperative activities are stored. A list of such servers including their exact addresses is managed in CAGISTrans. This list can be provided by a system agent to external agents. Note that, as mentioned above, the distribution of document servers is intended to be transparent to the users. Thus, the use of agents here aims at facilitating such a transparency. Finally, external agents may "poll" notification information that is useful for awareness purposes. This means that some of the awareness presentation services can be presented by system agents to external agents, which could be again "forwarded" to users.

An external agent asks the facilitator whether there are system agents that can offer the services it needs. If the service request matches one of those being advertised, the facilitator will provide the external agent with the address and the symbolic name of an appropriate system agent. Hereafter, the two agents will continue their dialogue until the external agent has accomplished its tasks.

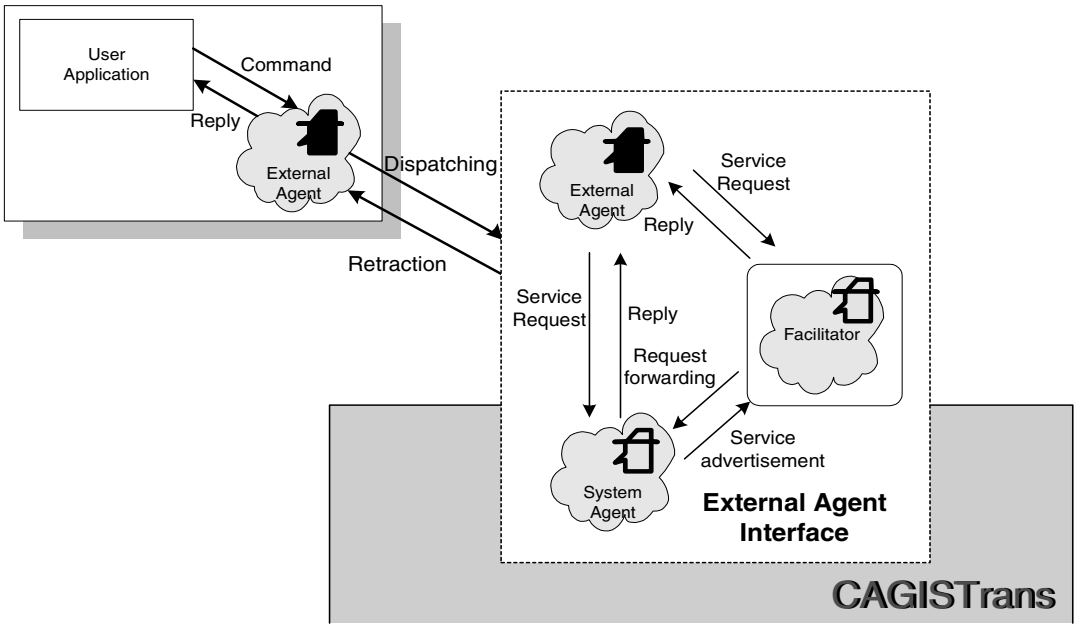


Figure 9.5 Interaction between an agent and the external agent interface components.

In addition to being KQML agents, external agents may also be aglets. Since our system agents are aglets, unlike a KQML agent, an (external) aglet may communicate directly with a system agent, exchanging messages with it until its service requests and corresponding tasks are accomplished. Thus, intervention by the facilitator is not necessary. Here, communication between two aglets follows the agent transport protocol (ATP) for message passing. See (Lange and Oshima 1998) for an overview of the ATP.

9.3. Towards agent-based groupware with transactional support

Since our CAGISTrans prototype is agent based it opens several opportunities for use in the context of agent oriented applications. An example is in the development of an agent-based groupware system with transactional support. Although developing agent-based groupware is beyond the scope of this work, the following discussion could still be useful considering the applicability of a CAGISTrans system.

Parts of this section have appeared as (Ramampiaro et al. 1999), based on a joint work with other CAGIS colleagues. The parts included in this section are mainly written by the present author.

9.3.1. Agent-based groupware model

Agents have increasingly been used in development of groupware systems. The fact that they can handle sophisticated interactions with other partners, thus being cooperative entities, has made them useful regarding support for cooperative activities. For this reason, the transactional support provided should also take agents into account as "partners" of the cooperative effort. This introduces additional challenges as the cooperation is thus not only between two or more humans, but also between humans and agents as well as between two or more agents. Because of this, in agent-based groupware, transactional support must play a role that is even more crucial than in traditional groupware, where only the cooperation among humans is primarily considered. Taking this into account, a possible interaction model that we have outlined can be as depicted in Figure 9.6 (Ramampiaro et al. 1999).

As shown in Figure 9.6, there are three levels of cooperation. The highest level is where the human users cooperate and interact with each other, possibly through a computer. Here, the cooperation patterns are represented by dashed arrows, while solid arrows correspond to generic non-cooperative interactions such as database accesses etc. This level is called the *human level*. The next level

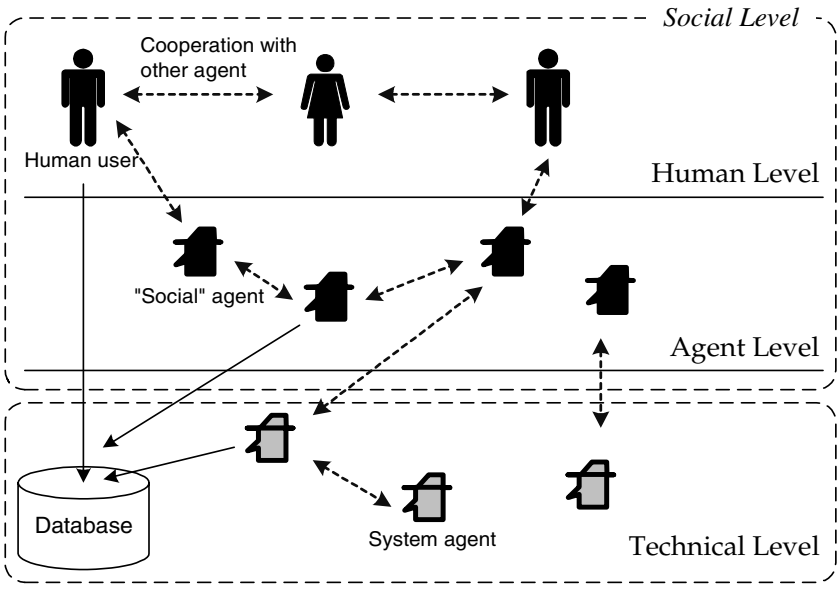


Figure 9.6 Illustration of a possible agent-based groupware model.

is called the *agent level*, where the main components are agents. This is where the agents cooperate and/or interact to perform tasks either on behalf of the human users or on their own accord. These interactions may be triggered as a result of the user interactions at the higher level. Finally, the lowest level in the model is the *technical level*. It contains the underlying computer system (applications, tools, and documents) and the resource management system – e.g., a database. The first two levels are together identified as the *social level* to focus on the social abilities characterising the participants to any cooperative effort.

Based on the three levels identified, the following concepts must be addressed in the transaction support:

- *Agents*, including:
  - *human agents* – i.e., the end-users of the system and the main participants in the cooperative effort. These can be either individuals or groups. Human agents are the agents with the highest degree of freedom, bound only by the organisational context in which they act.
  - *"social" agents* – i.e., the agents that have goals to support the cooperative effort that takes place at the human level and that, thanks to their social ability, play an active role in the effort itself. Some of these agents act on behalf of human agents (user agents), who can delegate them duties and responsibilities, binding their freedom. The duties and responsibilities of the other social agents are instead determined

by the role they play in the cooperative effort (possibly with the mediation of users).

– *system agents* – i.e., the agents that are functional to the system, for example, facilitating communication among agents and keeping track of the system status. Though these agents can cooperate with others in order to have their work done, they are not directly involved in supporting the cooperative effort at the social level. Their duties and responsibilities are determined by the system configuration.

- *Objects* – i.e., all the components of the system (or external components with which the system interacts) that cannot be classified as agents. These include databases, file systems, document systems, and the likes – e.g., the database in Figure 9.6.

In the following section, we outline some of the challenges that have to be faced in order to provide transactional support for agent-based groupware systems that are able to support distributed cooperative teams, such as software development teams. We anticipate that such teams involve cooperation not only among humans, but also among all the other types of agents defined above, thus introducing new challenges for transactional support.

9.3.2. The challenges

First, it is necessary to identify which types of transaction mechanisms should be used at which level. For example, what is the meaning of preserving consistency at the three levels, and how can it be supported? It is evident that most of the terms used traditionally in the context of transactions take a different meaning when fully considering the social level, and this requires a complete rethinking of the support that can be provided. However, by providing our user-oriented mechanisms such as awareness, user managed locks and workspace management, our CAGISTrans system might address this challenge. We recall that our awareness service makes it possible to provide users with knowledge about events that may be relevant for their actions. This will, for instance, help them foresee potential conflicts such that they can avoid them by, for instance, using our user-controlled and collaborative locks (see Section 7.2.4.1). Hence, this could permit them to manage consistency at the user level – i.e., consistency according to users' view. Further, CAGISTrans provides an infrastructure supporting agent interactions. This allows agents to cooperate with other agents following specific cooperation "patterns" and policies that are specified by the user. Hence, our prototype allows a user to specify correctness criteria, and all external agents using the provided CAGISTrans

services must obey criteria corresponding to their environments. Through this, our CAGISTrans prototype may address the consistency management challenge at the agent level.

The second challenge is to find a constructive way to define and modify policies at the different levels that also suit different situations. This requires the definition of a language for describing policies. This language must be "easy enough" to be used by human agents and, at the same time, "formal enough" to be used by software agents. CAGISTrans allows specification of applied policies through the transaction characteristics specification. Our use of XML addresses the need for a language that a user may understand, since most users are familiar with basic ideas behind XML. Further, there is a mapping between XML and KQML (or ATP messaging for aglets) – as discussed in Section 9.2.3. This makes it possible to propagate policies specified for user interactions to agents at a lower level.

The third challenge is to find a constructive and effective way to delegate responsibility and duties from one agent to another. This issue has been widely investigated in traditional groupware systems, where the delegation can be from one user to another, or from the user to the system. In agent-based systems, the scenario gets more complicated because delegation can involve all the types of agents identified in the previous section. Delegation of responsibilities and duties also implies the (often implicit) delegation of rights such as for accessing documents or modifying predefined patterns of actions. How can the transactional framework support this delegation when it is done both explicitly and implicitly or as a consequence of another delegation? CAGISTrans focuses on delegation of responsibilities among transactions. This challenge can thus be addressed if all actions performed by agents as well as human users can be captured and represented as transaction operations. More specifically, agent actions that are modelled as transactions can use the dynamic restructuring feature of our framework to delegate tasks at runtime. In this way, delegation of responsibilities, such as delegation of locks on objects or transaction operations, can be accomplished as described in Section 7.3.1.2. Nevertheless, helping human users to avoid them from incorrectly altering the delegation of responsibilities among transactions, and thus improperly breaking the barriers among transactions, is an important challenge that still must be taken into account.

Related to the previous challenge, migration and merging of capabilities or rights are not straightforward and need special consideration. How could we, for instance, manage the effect of delegating responsibility at the human level to the agent level? In other words, what is the effect of this on the human

and the user agents' rights? For example, a human agent delegates the responsibility for retrieving some data from a database or modification of a plan of action to his/her user agent. What happens if the human agent and the user agent access the information at the same time? Have the human agent's rights migrated to the user agent? Our access control policy considers a human user as having privileges superior to agents. Therefore, although rights might have been migrated to agents, the user can still override agents' access rights. This is also necessary for security reasons, hence preventing events such as virus problems.

Exception handling must be considered at the three levels, and it represents a challenge that deserves special consideration. As a first step, it is essential to understand the meaning of the term "exception". At the technical level, exception is mostly referred to as "errors or faults". This is, however, not always true at the social level, where exceptions are related to unexpected events that can possibly have positive effects. For a discussion, see (Saastamoinen 1995). In addition to this, exceptions cannot be completely foreseen a priori, at least at the social level. What is the support that is needed? In addition, how do exceptions at one level influence the other levels? Exceptions such as concurrency anomalies or failure anomalies can be managed readily through traditional concurrency and recovery protocols (Bernstein et. al., 1987). At higher levels, the picture becomes more complicated. However, by providing a flexible abort management scheme, allowing users to freely cancel actions as a result of, for example, completion of other actions, we may manage exceptions in a "user-friendly" way. Further, our support for dynamic re-specification of correctness constraints, combined with workspace support, allows users to adjust the level of cooperation as needed. So, for instance, in a cooperation situation, if someone is stuck, and needs help from others, our system allows "on-line" relaxation of criteria such that cooperation is possible. Such a type of event – the fact that someone can get help from others in an unplanned way – might be regarded as "positive exceptions".

Chapter 10
Integration with the CAGIS Environment

The work in this thesis was conducted in the context of the CAGIS project (see Section 1.5). The CAGIS environment consists of three main components.

1. A system for handling distributed documents and document understanding – also called *document models and tools.*
2. A system for supporting cooperative processes in a distributed environment – also called *process models and tools*.
3. A system for supporting transaction management for shared, distributed resources – also called *transaction models and tools*.

Component 3 has been presented in this thesis, now Sections 10.1 and 10.2 will describe the efforts in the two other research areas in more detail. Section 10.3 presents a test scenario that we will use to outline the integration of our CAGISTrans framework with the other CAGIS components in Section 10.4. Finally in Section 10.5, we discuss our approach.

Parts of this chapter have appeared as (Ramampiaro et al. 2000), based on joint work with colleagues in the CAGIS projects. Hence, Section 10.1 and Section 10.2 came mainly from colleagues (with some minor modifications), the scenario in Section 10.3 was a truly joint composition, while the material on transactions in Section 10.4 and the main contents of Section 10.5 were written by the present author.

10.1. Document models and tools

The development of a system for handling distributed documents and document understanding in CAGIS was motivated by the fact that documents published on the Web have to be organised, classified and described to facilitate later retrieval and use. Semantic classification – i.e., representation of document contents – is one of the most challenging tasks. This is usually done using a mixture of text-analysis methods, a carefully defined (or controlled) vocabulary or ontology, and a scheme for applying this vocabulary when describing a document. The CAGIS document model toolset is aimed at helping users in a project group to do this semi-automatically, by way of a domain model expressed in a conceptual modelling language and by using text analysis tools as an interface to perform the actual classification and search. This means that one may use *a conceptual model* as a basis for creating meta-data descriptions (see Figure 10.1). These meta-data descriptions may then be accessed through a java-model viewer that enables search and browsing of documents through a standard Web browser environment.

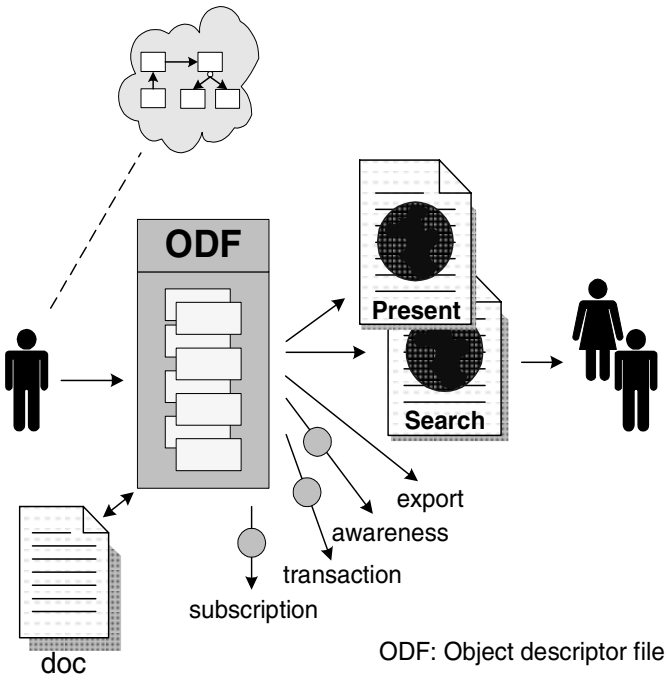


Figure 10.1 Conceptual modelling for meta-data descriptions.

Fundamental to this approach is the use of a conceptual modelling language to define and visualise the domain specific vocabulary to be used in the classification and retrieval process. Conceptual modelling languages contain the formal basis that is necessary to define a proper ontology. At the same time,

they offer a visual representation that allows users to take part in the modelling, and read and explore documents by interacting directly with the models. The conceptual modelling language may thus be used throughout the entire process of classifying and retrieving documents on the Web. The modelling language used in CAGIS is the *referent model language* (Sølvberg 1999), an ER-like language with strong abstraction mechanisms and a sound formal basis.

Document handling in CAGIS may be described as a three-step process (Brasethvik and Gulla 1999): *Domain Model Construction*, *Document Classification* and *Browsing and Retrieval* – outlined below.

10.1.1. Domain Model Construction

Conceptual modelling is mainly a manual process. However, each domain model must be related to the text of the documents to be classified, hence a textual analysis tool is used as input for the modelling. A reference set of documents from the domain is processed through a word frequency analysis tool, which produces a list of high frequency terms as input candidates for the actual conceptual modelling task. This is a manual and cooperative task performed by a selected set of users. Concepts are carefully selected, related to each other and given a textual definition. In order to prepare the finished domain model for later document classification, lexical linguistic information is added to the model. This means that the model is enhanced by adding a term-list for each of the concepts in the model. The term-list is a list of synonyms, instances and conjugations for each concept that will be used later in the classification of a particular document.

10.1.2. Document Classification

Documents are classified by selecting domain model fragments that reflect the document content. This is performed semi-automatically by matching the document text against the term-lists for each of the concepts in the model. Concepts found in the document are then shown to the user as a selection in a graphical model viewer, and the user may manually refine the classification by selecting and deselecting concepts and relations. When the user is satisfied, the selected model fragment is translated into XML and is stored as an Object Descriptor File (ODF). The user also has to provide a selected set of properties for the document, such as its author, title etc. These attributes are also stored within the ODF.

10.1.3. Browsing and Retrieval

In order to retrieve documents, the users enter a natural language query

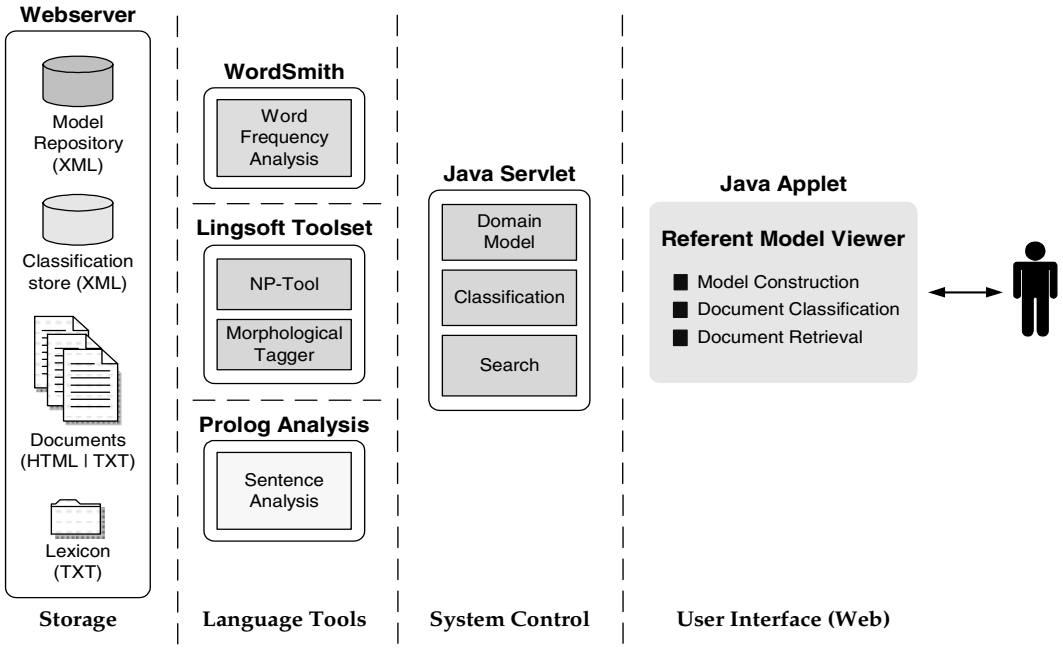


Figure 10.2 Overview of the system architecture.

phrase which is matched against the conceptual model in a similar way to the classification process. The domain model concepts found in this search phrase (if any) are extracted and used to search the stored document descriptions. Users may then refine their search by interacting with the model. Matching documents are presented as a list in a Web-browser interface. This system also has an enhanced "document reader", which means that when reading a document, all the terms in the document that matched a model concept are marked as a hyperlink pointing to the definition the model concept.

The layered architecture of the CAGIS document tool is shown in Figure 10.2. The main parts of the system are the Web-enabled user interface and a set of servlets running on a standard Web server.

- The user interface is centred around a Java-based *Referent-model viewer*. As mentioned, users may interact with the model, explore concept definitions and relations, and then use the viewer directly in order to perform both classification and retrieval.

- The *Java servlets* define the overall functionality of the system. They are invoked from the model viewer and coordinate the linguistic tools incorporated in the system.

- At an *"intermediary" layer*, between the servlets and the Web server, there are several linguistic tools that analyse natural language phrases and give the necessary input to construct domain vocabularies and

classify and retrieve documents. The word frequency analyser from WordSmith is a commercially available application for counting word frequencies in documents and producing various statistical analyses. A Finnish company, Lingsoft, has two tools for analysing nominal phrases and tagging sentences needed for the classification and retrieval of documents. A smaller Prolog application for analysing relations between concepts in a sentence is being developed internally at the Norwegian University of Science and Technology.

- Finally, the documents and their classifications are stored as files at the Web server in HTML/TXT and XML format respectively. The domain model is also stored in XML and must be maintained separately. The linguistic tools use lexical information that is partly stored in the model XML file or as a separate lexicon TXT file and partly integrated within the tools themselves.

A more detailed presentation of this approach and the system is given in (Brasethvik and Gulla 1999, Brasethvik and Gulla 2000).

10.2. Process models and tools

The development of the system for supporting cooperative work processes in CAGIS was motivated by the need to structure – i.e., plan, coordinate and organise – work performed across a distributed network – i.e., the Web.

A prototype of a process centred environment (PCE) has been developed to give process support to distributed, cooperative processes in CAGIS. The CAGIS PCE consists of three main components (Wang 2001): (1) Workflow system supporting distributed mobile processes, (2) Software Agents to support dynamic, cooperative processes and (3) Agent-Workflow GlueServer – outlined below.

10.2.1. Workflow system supporting distributed mobile processes

The CAGIS workflow system is used to model simple, repeatable workflow processes, and the system offers an agenda-browser for the end users. The workflow system allows an instantiated workflow model to be distributed as several process fragments on different workspaces. One benefit of this is the possibility to adapt the workflow to local environmental conditions. The workflow model instances are defined as XML-files distributed over several workspaces, and can be modified by the owner of the workspace. The ability to move and change workflow instances during enactment, can be used for reallocating activities, dealing with exceptions – i.e., when someone responsible for a

particular activity performs unexpected actions, and delegating work. The workflow system is implemented in Perl, providing a CGI-interface through a Web server. A more detailed description is provided in (Wang 1999, Wang 2000).

The Process Modelling Language (PML) for the workflow system defines a process as a set of activities that can have mutual pre-order relationships specified in XML syntax. This means that an *activity* can specify a set of *pre-links* identifying the activities to be executed before, and *post-links* identifying the activities to be executed after the current activity. The pre- and post-links are written as URLs. Therefore they allow a process to be distributed over several workspaces. Every activity definition specifies a code part (a script). This code part is expressed in HTML, and can be used to simply present information, to specify a user input through a form, or to start a Java-applet. The term *process fragment* is used to name a group of activities in a workspace as part of the whole process. A process fragment is specified by a name, a workspace (location), and a list of references to activities.

10.2.2. Software agents to support dynamic, cooperative processes

While the workflow system described above takes care of simple, repeatable processes, agents are used to support more cooperative and dynamic processes. Agents are typically responsible for inter-group activities such as brainstorming, voting, negotiation activities – e.g., concerning resource allocation, coordination – e.g., of artefacts and workflow elements between workspaces, market support – e.g., agents as buyer and sellers of services, etc. The CAGIS multi-agent architecture consists of four main elements (Wang 2001):

- *Agents*. An agent is set up to achieve a modest goal, characterised by autonomy, interaction, reactivity to environment, as well as pro-activeness. There are three main types of agents: (1) *Work agents* to assist in local production activities, (2) *Interaction agents* to assist with cooperative work between workspaces, and (3) *System agents* to give system support to other agents. Interaction agents are mobile, while system and work agents are stationary.

- *Agent Meeting Place (AMP)*. AMPs are where agents meet and interact. AMPs support agents in providing efficient inter-agent communication. There can be different types of AMPs for different purposes. Each AMP will have a defined ontology (the framework described in Section 10.1 can be used here), which the agents have to follow. We can perceive special AMPs for negotiation, coordination, information exchange, selling and buying services etc.

- *Workspaces.* A workspace is a temporary container for relevant data (artefacts, models etc.) in a suitable format to be accessed by tools, together with the processing (work) tools. Files stored in a repository can be checked in and out to a workspace.

- *Repositories.* Repositories can be global, local, or distributed, and provide persistent storage of data. Experience bases are one specific type of repository that we can use in our multi-agent architecture to support community memory.

The multi-agent architecture is implemented in Java, using IBM aglets software development kit (ASDK) to provide mobile agents. KQML is used for inter-agent communication, and ORBIX CORBA is used to offer communication to other applications and other agent systems. A more detailed description of the multi-agent architecture can be found in (Wang 2001).

10.2.3. Agent-Workflow GlueServer

The Agent-Workflow GlueServer provides interaction between the workflow system and the multi-agent system. A *glue model* in XML defines the relationship between workflow elements and agents. The *GlueServer* provides services for a workflow activity – e.g., triggering an agent. An agent may also initiate a workflow activity through this GlueServer. It is implemented in Java. ORBIX CORBA is used to facilitate communication with the agent system and workflow systems. The GlueServer is discussed in more detailed in (Wang 2001).

Figure 10.3 illustrates typical interactions among different components of

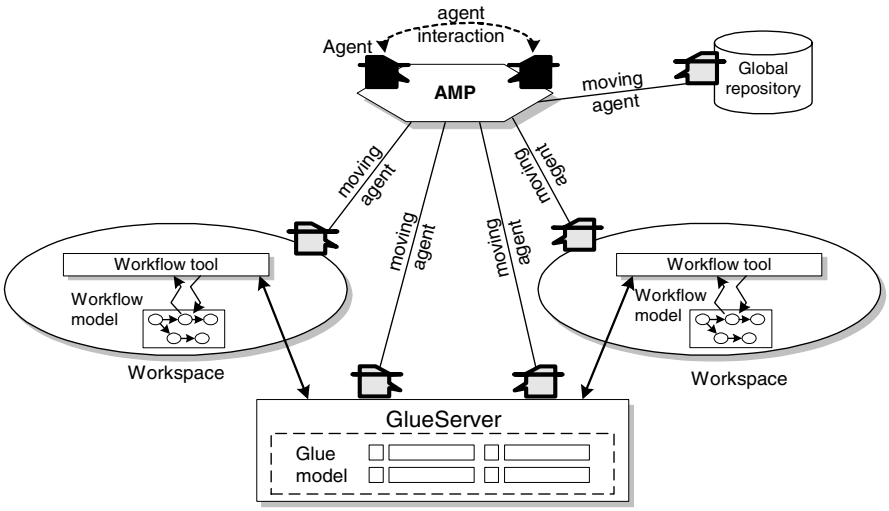


Figure 10.3 The CAGIS Process Centred Environment.

the CAGIS PCE. Here, each workspace executes a workflow tool (engine) using a local workflow model. In practice, this workflow tool may be shared, and local workflow models in two different workspaces may have relationships. The figure also illustrates two different ways in which an agent may interact with a workspace. It may interact directly with the user in the workspaces through a graphical user interface used to configure and interact with the agents. Alternatively, all interaction with agents can go through the GlueServer and the workflow tool. Finally, access to repositories can be achieved through workspaces directly, or agents can be used to access them.

10.3. A practical test scenario: conference management

Again, to best illustrate the use of the three CAGIS components, we have adopted a conference organising scenario based on that presented in (Olle et al. 1982).

A conference organisation process may consist of several activities, starting with a Programme Chair *planning* and *announcing* the conference. Thereafter, people may contribute to the conference by submitting their papers. Members of the programme committee (PC) *register* submitted papers as well as information about the authors. Then, *referees* for each paper are *chosen* based on their expertise, and the *paper review* starts. The PC members *collect the results* from the review, and an electronic review meeting is held to *select papers* for the conference. Accepted papers are *grouped into sessions* and a final programme, including a time-table for conference sessions, is produced.

Here, we focus on the last main activity – i.e., *group accepted papers into sessions*. Figure 10.4 illustrates the two main sub-activities of this activity.

10.3.1. Suggest sessions

The Programme Chair is responsible for this activity, which can be decomposed into the following process steps:

(1) Matching all papers against a document model, defining terms and expressions, and the relationships among them for the research domain.

(2) Suggesting a session division according to subjects.

(3) Creating a preliminary session schedule.

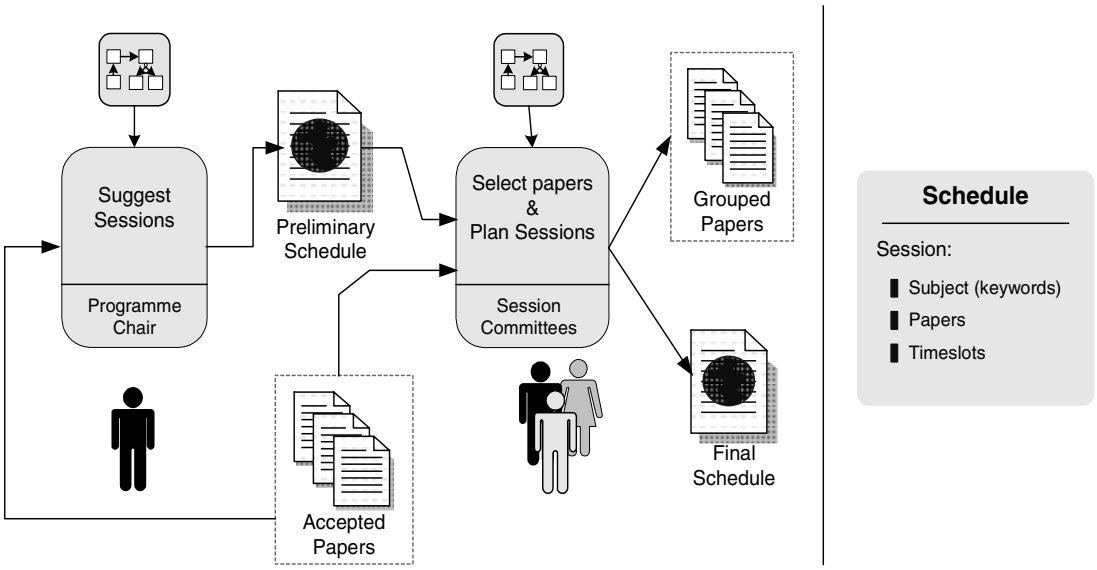


Figure 10.4 Grouping of Accepted Papers into Sessions.

(4) Setting up session committees (from programme committee members), one for each session.

10.3.2. Select papers and plan sessions

Each member of a session committee is responsible for selecting papers and planing a session. This activity consists of the following process steps:

(1) Determining session subject and goals. An *initial* session description contains the session subject and goals.

(2) Checking papers for session. Session committee members mark papers that are relevant for a session, showing their interest. Papers are marked *"possible"*.

(3) Allocating papers. If papers are marked by more than one session committee, these committees must negotiate about which session is going to get the paper. Papers finally allocated to a session will be marked *"taken"*.

(4) Checking timeslot for session. Each session committee marks a timeslot for the session.

(5) Allocating sessions. Sessions that have the same timeslot are subject to negotiation. When all sessions are allocated, the result will be added to

the session description. The session description will now have the state *"final"*.

(6) Publishing session description: Each session committee will publish their session description to the other session committees and programme chair.

Here, we assume that the programme committee members will be distributed across different locations, and that the organisation of the conference will be done through computer interaction – i.e., without any physical meetings.

10.4. The CAGIS environment applied to the scenario

This section outlines how our CAGIS tools and models – supporting documents, processes and transactions – can be applied to the scenario described above. The CAGIS environment for this scenario is shown in Figure 10.5.

The two main activities that we are focusing on – i.e., suggest sessions and select papers and plan sessions – are executed by the programme chair and the session committees respectively. In our solution we have chosen to model

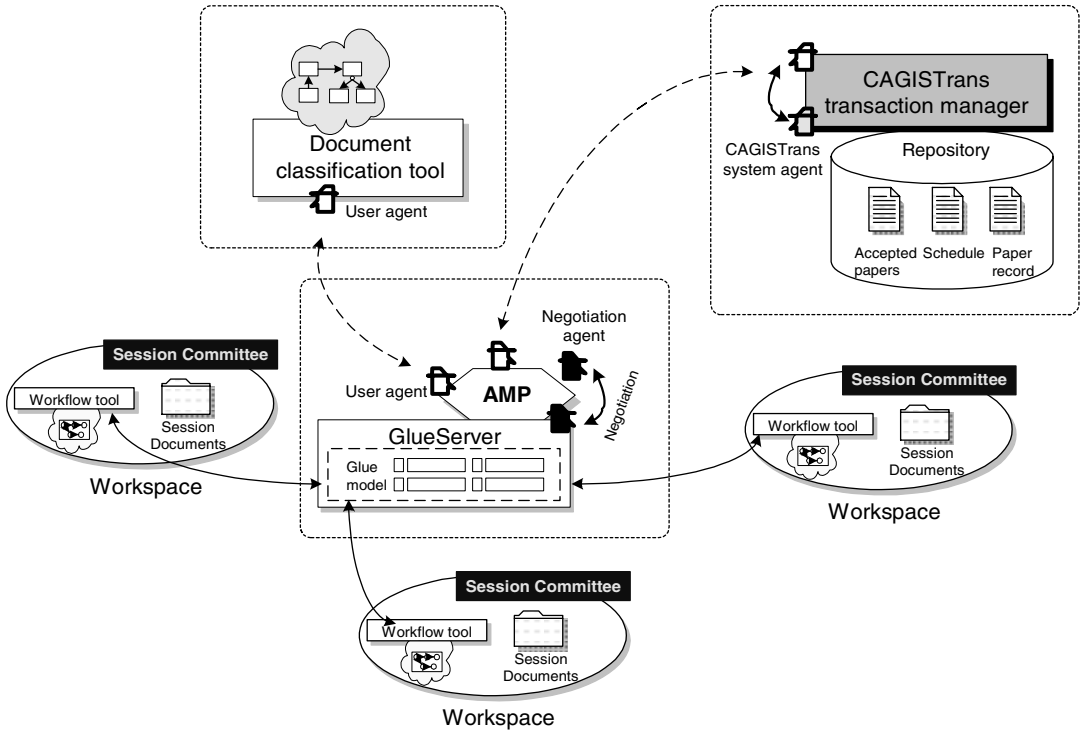


Figure 10.5 The CAGIS framework applied to the scenario.

the scenario using one workspace for the programme chair and one workspace for each session committee. Each workspace has a local process defined in a process model, and a workflow tool that enacts this process model. The process models are defined according to the process steps defined for suggest sessions and select papers and plan sessions as described in Section 10.3.

The programme chair's first task is to classify all papers according to their themes. This is done by using the *document classification tool* to match all papers against the domain model. The domain model defines the vocabulary of keywords, extracted from the preliminary conference topics and from the submitted papers. The matching of papers against the domain model is visualised in the document model viewer, thus illustrating how the papers are thematically related to the concepts in the domain model. The programme chair may interact with the model viewer to achieve a proper subject division of papers.

The *workflow tool* will here notify the *GlueServer*, initialising a *document agent* to access the documents through the document servlet. Next, the workflow tool provides the programme chair with necessary documents and tools for creating a preliminary session schedule and setting up session committees. When session committees are selected among PC members, the session committee members will be notified by e-mail describing what session committee to attend and what workspace to access.

The session committees will then start working in their workspaces according to the process model enacted by the workflow tool (recall that all conference organising work is to be done with distributed computers). First, they have to determine session topics and goals. For this, the workflow tool notifies the GlueServer that initialises *brainstorming agents* for each session committee member. The result from this brainstorming process is an initial session description written by a session chair. Further, the session committees select papers that are appropriate to their session. Here, the workflow tool notifies the GlueServer to initialise *paper select agents*. The paper select agents will retrieve information about available papers, and let the session committees mark interest in papers. The paper select agents will then mark papers in the *paper record* in the repository (see Figure 10.5). The result from marking papers is returned to the GlueServer. In cases where a paper is marked by several session committees, negotiation agents are initiated to negotiate. This determines which session will get the paper. If this negotiation process goes into a deadlock, the programme chair will be notified, who will then make a final decision. After all papers are marked, each session committee chooses a timeslot for the session. This process is similar to paper selection, but *session selection agents* are used instead. When all session committees have selected their timeslots, the final ses-

sion description is published to all participants, and the final conference programme can be produced.

The role of the *CAGISTrans transaction manager* in this scenario is to ensure the integrity of the document in the repository, and to ensure that agents always leave the system in an acceptable state. Based on the description above, session committees share both the schedule document and the paper record. Hence, conflicts are likely to occur. Our CAGISTrans system provides several possibilities to resolve these conflicts. First, one may exploit the awareness feature (see Section 7.2.4) to notify each involved party so that they may prevent potential conflicts. Alternatively, one may use a user controlled lock (see Section 7.2.4) for each access, thus prohibiting others from seeing any changes until a related process is finished. However, this is usually unacceptable, since it might delay the session arrangement process. Another possibility is then to permit read accesses, thus allowing other committees to see the intermediate changes. This can be achieved by defining permits on data that were checked out from the public repositories to committee workspaces. This will ease each committee's decision process. A third solution is to permit simultaneous updates (i.e., write/write conflict). However, achieving consistency is subject to negotiation determining which update results should be regarded as final – i.e., consistent in the eyes of all involved parties.

Further, tasks to support the session selection process are assigned to agents. This also involves document accesses. To allow our transaction management system to ensure consistency, all agent operations involving repository access are managed as part of transaction executions. This also ensures that conflicting document access is managed properly. Moreover, suppose that a transaction consisting of several agent operations is initiated by the GlueServer. Then, assume that one of the involved agents fails, for example, while selecting papers from the paper record. Using traditional ACID transactions, this would cause a global rollback, that would discard all changes made so far and kill all associated agents. However, if a lot of effort has already been invested, restarting all tasks from scratch could be unacceptable. To cope with this, we model each agent operation as a subtransaction of that executed by the GlueServer. Therefore, instead of aborting the transaction and killing all involved agents, the transaction manager allows the failing agent to rollback its tasks and restart if necessary. Other agents that are not directly affected may proceed as normal.

Agents initialised by the GlueServer are seen as external agents from the viewpoint of the CAGISTrans agents. Thus, referring to Section 9.2.3, when agents from session committee workspaces access the global repository they

may do this through the CAGISTrans external agent interface. This interface will then provide services according to the agents' requests.

10.5. Summary and discussion

This chapter has presented the use of our CAGISTrans prototype as part of the CAGIS toolset. This was done using a conference organisation scenario. It has demonstrated the CAGIS toolset consisting of a set of separate tools which can be used together to provide support for cooperative work across the Web. The three major components of CAGIS presented and discussed in this chapter have been developed within CAGIS. Each of these tools has been implemented in true Web style – i.e. they are built around a standard Web server and use XML as a data storage and interchange format. Fundamental to our CAGIS approach is the development of tools that may be configured according to the actual situation and use. The *workflow tool* allows creation of individual workspaces supporting the execution of workflows. In addition, the workflow tool offers the ability to enact parts of the process model supported by workspaces. The *document classification tool* uses a domain specific vocabulary – i.e., a domain model – to help users classify and search documents. And, our *CAGISTrans* – i.e., the *transaction tool* – offers support for the specification and execution of customised application-specific transaction models. A *GlueServer* was used to bind the individual tools together. The GlueServer configures a set of software agents that can interact with different CAGIS tools. As part of the GlueServer, there is a *glue model* that defines the relations between individual workflow elements residing in different workspaces and the software agents used to access the individual tools.

Our CAGIS environment is not only applicable to conference organisation processes. It is our allegation that the CAGIS environment can be used to support diverse processes involving people working together, where both people and information are distributed. Examples of such processes include cooperative software engineering processes, distributed educational processes, distributed organising processes, processes of selling and buying merchandise on the Web etc. All these are characterised by distribution of people and information, and require people to interact and cooperate to reach a goal.

An important result of integrating individual tools in a CAGIS environment is that we may benefit from enhanced functionality of each specific CAGIS tool. From our point of view, this means that we may exploit the existence of a document model to define document integrity. This may, for instance, be used to determine acceptable final results of transaction executions. Further, a document model may be used to specify the ontology of the agents in our ex-

ternal agent interface. An ontology here is a static document that can be used to specify diverse items – e.g., the agent communication language to be used.

The workflow tool can be used to model repeated, well-formed tasks. From our perspective, this means that for activities consisting of repetitive processes, the workflow tool can be used to capture actions that are necessary to execute to achieve a "goal". Hence, we may facilitate the specification of our constraints determined by demands in our CAGISTrans framework. In conclusion, the possibility to model activities using the workflow tool may be useful.

Seen from a document and workflow modelling point of view, the existence of our transactional support is also useful. For example, document models are stored on databases or Web servers. We may then use transactions when updating these models to ensure their consistency. Further, Wang (2000b) discusses the use of the transaction models and tools to offer a way of managing consistency of changes to workflow models. Both document and workflow modelling can benefit from the flexible, advanced transactional support provided by our CAGISTrans framework.

In conclusion, our CAGIS environment offers a selection of tools which can be exploited in different combinations to give specific useful support. What remains to be done is to further develop guidelines facilitating the designation of tools suitable for specific scenarios. This will make our CAGIS tools even more efficient and user-friendly.

Chapter 11
Discussion and Evaluation

This chapter discusses (Section 11.1) and evaluates (Section 11.2) my work in the CAGISTrans part of the project and compares it with related work (Section 11.3).

11.1. Discussion

This discussion will focus on issues concerning our CAGISTrans transactional framework. These issues include discussing the trade-off between user intervention and system transparency, addressing performance issues, using agents as development platforms and some implementation issues – outlined below.

11.1.1. User intervention vs. system transparency

The trade-off between system transparency and user intervention is a key issue worth discussion. From a CSCW perspective, too much system management and too little user control is unacceptable. Users might then get the feeling of losing the whole picture of what is going on, which is against the philosophy of CSCW. As a result, most CSCW applications rely on social interactions to handle concurrency (see Section 3.1). But, from a database point of view, this is unacceptable as data consistency cannot be guaranteed. Thus, this has given rise to the need for a sensible trade-off. The specific combination of user-managed specifications and system-based control in our CAGISTrans framework may give an acceptable result. Our framework has been designed to allow users to specify appropriate transaction models based on application needs, and have the system do the validation, management and control based on this. However, the cost still to be paid is that users may be required to have

a level of expertise above what can be expected in an average user. This triggers the necessity of developing a graphical user interface to make our system more user friendly. This will, for instance, relieve users from coding transaction models in XML themselves.

11.1.2. Performance issues

Another important issue is performance such as transaction throughput. Traditionally, high transaction throughput has been one of the main requirements for transaction processing systems. The discussion in Section 7.3 indicates that our framework's execution of advanced transactions may introduce some overhead. For example, several execution "parameters" have to be in place before a transaction is executed. Further, introduction of new constraints may also imply some control and management tasks, which could slow down the overall speed. However, transactions supporting cooperative work exist for long periods of time. Therefore, they may be more sensitive to response time performance than system throughput. Hence, some extra time needed for validation, management and control purposes would be less critical in the global picture. For this reason, we have primarily emphasised provision of flexible support for cooperative activities rather than optimization of the transaction speed itself. Nevertheless, since users would normally expect a minimal response time, optimising the overall system performance is an important issue that deserves further development. Thus, this will be a significant subject for further research.

Following this line of arguments, another performance issue worth discussion is our use of Java[1] as the implementation platform. Currently, the most significant weakness of Java is its moderate performance, at least compared with C/C++. However, Java is an interpreted object-oriented language with a capability of being executed on heterogeneous environments. In addition, Java programs are inherently capable of being transported over the network and executed at the remote system. Due to our distribution and heterogeneity requirements, the choice of Java was a simple one to make. It is nonetheless important to note that Java's moderate performance can be improved by making Java code executable instead of interpretable. In fact, to date, so-called Java just in time compilers (JIT)[2] are available to help improve the speed of "executing" Java applications.

1. See http://java.sun.com or http://javasoft.sun.com.
2. See http://www.sun.com/solaris/jit.

11.1.3. The use of agents

The CAGISTrans prototype has been developed based on an agent framework. There are several reasons for this. First, a conclusion from the above discussion is that system throughput is not as critical as in traditional transaction processing systems. However, since a CAGISTrans system operates in a distributed environment, its performance will also be affected by network latency. In the long run, this is unacceptable. Our use of mobile agents such as aglets was motivated by the need to cope with this problem. For example, by exploiting the mobility aspect of aglets, our CAGISTrans system allows interaction between an external agent and a system agent to appear at the CAGISTrans system site. When the interaction is accomplished, the external agent returns to its application with the results. On the other hand, if we had used other distributed technologies such as CORBA[1] and DCOM[2], the application would have had to interact with a CAGISTrans system directly through the network. Thus, the system would very much have been dependent on network speed.

Moreover, the CAGISTrans framework is intended to provide flexible transaction models for cooperative work. This implies that the technology platform used must support such transaction models too. CORBA provides a transaction service called OTS – object transaction service (Object Management Group 1998). However, this service only supports the ACID model, which is too rigid for our purposes (see Section 5.3). On the other hand, mobile agents such as aglets are not bound to any specific transaction model. This has allowed us to implement our framework with the ASDK. Hence, an explicit contribution to agent research of our work ought to be the provision of adaptable transactional services with agents.

Note that there are cases where current agent technology alone does not solve all platform-related problems. It has been pointed out that there is no widely accepted standard agent platform. Therefore, the issue of interoperability is still a challenge. To cope with this, we have chosen to combine the use of aglets with a more standard technology – i.e., Java RMI[3]. This has enabled us to exploit both the mobility aspect of aglets and the interoperability aspect of Java RMI.

11.1.4. Implementation issues

The external agent interface in our prototype has been implemented with an aglet server, intended to handle several external agents (see Section 9.2.3).

1. See http://www.corba.org.
2. See http://www.microsoft.com/tech/dcom.asp.
3. See http://java.sun.com/j2se/1.3/docs/guide/rmi.

However, our experience with the latest version of ASDK is that an aglet server is not able to deal with more than one request at a time. Because of this, an aglet server may become an additional performance bottleneck. A solution that we are currently working on is extending the *CoopInterface* with a capability that can handle several simultaneous requests from external agents. As part of this, we are developing a host aglet that makes use of parallel threads, where the main thread is responsible for communicating with the facilitator.

Further, JKQML has proved to be a useful and convenient tool due to its simplicity and compatibility with ASDK. But, since JKQML has been used with aglets, KQML messages have to be relayed using ATP – i.e., the agent transport protocol (Lange and Oshima 1998). This implies that each mobile agent must carry all its messages along with its state. However, since almost all interaction among agents takes place on a specific host, this has not caused much difficulty. In fact, this extends the degree of distribution of our prototype. However, a certain limitation of JKQML is the way the *content language* is implemented. The content language supported is predefined. This has made it impossible for us to use XML to encode the content description. For this reason, future extensions of our prototype will benefit from further improvements concerning how KQML and XML can be used together.

11.2. Evaluation

In order to evaluate our CAGISTrans framework, the first issue is how it is able to meet the requirement for cooperative work. As part of this evaluation there is also an outline of how we were able to reach our goal through answering our research questions. Finally, the limitations of our CAGISTrans framework are presented.

11.2.1. Meeting cooperative work requirements

Fundamental to our framework is the provision of an explicit separation of design time and runtime specifications. This allows a transaction model designer to specify parts of a transaction model before transactions are executed, and modify other parts while transactions are in progress. This is crucial and necessary to cope with the dynamic nature of cooperative activities.

In addition, a set of requirements was set up in Section 5.3 as foundations for the development of our CAGISTrans framework. The following outlines how our framework meets these requirements (see also Table 11.1 for a summary).

Starting with TR1 and TR2, the *characteristics specification* is particularly

Requirements	How they are supported
TR1 Tailorable atomicity	Satisfied through our customised non-ACID requirements, including a fine grained abort management scheme.
TR2 Controlled sharing of intermediate results	Also met through our customised non-ACID requirements, including user-defined correctness constraints and workspace usage.
TR3 Support for open-ended execution	Fulfilled through our support for dynamic restructuring and user re-definable correctness constraints.
TR4 Distribution and heterogeneity	Supported through a CAGISTrans system exploiting the middleware principle, supporting advanced transactions.
TR5 Awareness	Satisfied through our integrated support of notification mechanisms.
TR6 Temporal data management	Met through our distinction between activity and system logs, where the activity log is available to users.
TR7 Access control	Fulfilled through our usage of separated workspace instances and support for access restrictions and authentications.

Table 11.1 Summary of how the CAGISTrans framework meets the requirements for Cooperative work.

aimed at providing the basic means to meet these requirements. First, it defines the main properties of transactions based on "non-ACID-requirements" – i.e., customised ACIDity. Thus, it allows a user to fit both atomicity and isolation to the needs of the application. Second, it determines suitable transaction structures, and defines the effects of one transaction on other transactions, in terms of transaction dependencies, each defined based on actual "non-ACID-properties". Such dependency management was used as a means to provide a fine grained abort management scheme, hence providing a fulfilment of TR1. Third, the characteristics specification allows a user to designate appropriate correctness criteria to be applied, in addition to determining the mechanisms and policies to be used to satisfy the designated correctness criteria. This serves as a means to allow a controlled sharing of resources, thus fulfilling TR2.

Next, the *execution specification* consists of elements that essentially affect the transactional behaviour. It is aimed at providing users with the possibility to refine a transaction model at runtime, and thus enabling open ended execution. As the execution specification allows dynamic restructuring of transactions, i.e, spawning of new transactions and delegation of responsibilities, this implies that initiated activities may be terminated without accomplishing all planned tasks. Furthermore, the ability to transfer responsibilities to other transactions allows a transaction to release data before it aborts. Thus, referring back to Section 5.3, such a dynamic restructuring support is useful in satisfying

TR3. Further, with open-ended execution of transactions, requirements may change. To meet new needs at runtime, CAGISTrans allows a user to refine the prevailing correctness constraints while his/her transactions are being executed. This aspect is also useful with respect to meeting TR3.

Motivated by the need to meet requirements TR2 and TR7, we decided to integrate the concept of *workspaces* into our framework, further emphasising the necessity of flexible but controlled sharing. Since a workspace allows groups of people to perform tasks together and share their data or artefacts, the existence of private and shared – i.e., group and public – workspaces enables individuals to alternately work in groups and perform activities in private. Thus, both TR2 and TR7 are satisfied through workspaces by allowing as much sharing as possible, but at the same time avoiding unintended and unauthorised manipulation of data. To be able to regulate the degree of sharing, we have accentuated the necessity of providing *group workspaces*, in addition to private and public workspaces (see Section 6.2). Such workspaces have made it possible to restrict sharing to members of a specific smaller group of people. Although this may result in some increase in management complexity, the main advantage is our ability to adjust the level of sharing according to – e.g., the number of involved individuals. New extended workspace access operations have been provided as part of the realisation of our workspace management. In particular, some of these operations have made it possible to release and update intermediate objects, which is useful for meeting TR2.

The CAGISTrans framework has been developed as middleware to fulfil the distribution and heterogeneity requirements – i.e., TR4. Our use of an agent platform has further satisfied this requirement. Moreover, CAGISTrans provides awareness support where cooperating parties can be notified of relevant information related to their activities. This has been realised through awareness agents and a status window. Hence, CAGISTrans meets TR5.

Further, our CAGISTrans framework distinguishes between system logs and activity logs. System logs are stored on stable storage and keep track of all actions that modify objects in a repository. They are used for recovery purposes and are transparent to the user. Activity logs, on the other hand, are intended for users, to let them know about the status of their ongoing tasks. This is particularly useful when they temporarily have to disconnect from their current activities. In this view, our framework addresses the temporal data management requirement TR6.

Finally, all users of the system must have a valid user name and password, and they must log on to the system before carrying out their activities. The system will thereafter provide a list of activities and corresponding work-

spaces that a user may join. Using this kind of control, the CAGISTrans system further addresses TR7.

11.2.2. Answering the research questions

Answering our research questions is a way to see whether we have achieved our goals with this research or not.

Starting with the main question – as stated in Section 1.3 – this thesis has attempted to answer:

> *How can one provide transactional support that is able to deal with the dynamic and heterogeneous properties of cooperative work?*

This question has been answered by an analysis of the problem, definition of a set of requirements, development of the CAGISTrans framework, and design and implementation of a CAGISTrans system. Our framework allows transaction models to be dynamically tailored in accordance with changes in the environment. It is also aimed at being portable and interoperable, allowing people to work together across different platforms and geographical boundaries. Our system design not only provides advanced transactional support but also allows utilisation of a wide variety of resource management systems, including legacy databases, Web servers, etc.

The above main question led to the definition of several subquestions, determining the development of the work:

> ***Q1*** ***Current situation:*** *are there efforts that have already attempted or answered the main question?*

Our state-of-the-art survey has revealed the existence of a variety of solutions to support selected aspects of cooperative work. From this we were able to explore the current situation. This has allowed us to identify features that are beneficial regarding the solution to our main problem. And, more importantly, it enabled us to discover what is missing and still in need of attention.

> ***Q2*** ***Requirements:*** *what is the nature of cooperative work and how can it be characterised? Then, what requirements does this impose on the transactional support to be provided?*

An analysis of cooperative work characteristics has been necessary to determine the requirements on transactions. This question was thus addressed by our ability to identify important requirements that must be addressed to provide adequate transactional support for cooperative work. The requirements were set up based on an analysis of characteristics of cooperative work.

Q3 ***Solution:*** *what foundations are necessary for the design of a system fulfilling these requirements? In other words, how can we meet these requirements for transactions?*

Our main conclusion from answering Q2 is that cooperative work is diverse in nature. Therefore, no single, fixed model (the approach used in many existing approaches) can provide satisfactory support. Rather, the trend has been on development of customisable transaction models or transactional frameworks. This makes it possible for transactions to be tailored to specific needs. So, to answer this question we have developed a transactional framework that allows users to tailor transaction models to suit specific applications, and at the same time enables them to refine models dynamically in accordance with changes in the environment.

Q4 ***Evaluation:*** *how well do our research results solve the problems, and how does the solution compare to previous work?*

Our CAGISTrans framework was shown to be able to support the basic features of dynamic transaction management, allowing users to specify models and have the system execute their transactions in a flexible but controlled manner. Our combination of customisable non-ACID-requirements, support for dynamic restructuring, dynamically adaptable user-defined correctness criteria, explicit support for applied policies, and integrated workspace management as a whole is unique compared to other existing transactional frameworks. A further comparison with other approaches is provided in Section 11.3.

11.2.3. Limitations of the CAGISTrans framework

There are several advantages with our framework. However there are several issues that are not addressed, either because they are beyond the scope of this work or because we were not able to do so, due to time and financial constraints.

- *Full ad-hoc support*. Team work often requires full ad-hoc support. This allows participants to perform work the way they want to, without the control of the system. Concurrency control, for example, is often left to social interactions. Seen from user's point of view, allowing such a possibility can be very useful. However, our system requires that either a user specifies correctness criteria and corresponding applied policies, or the underlying system provides them. This implies that users do not have the aforementioned freedom. Some may regard this as a shortcoming. We, however, argue that if full freedom is to be allowed, con-

sistency cannot be guaranteed. Moreover, this type of cooperative work support is covered by existing groupware applications. In conclusion, full ad-hoc support is beyond the scope of the CAGISTrans framework.

- *Stand-alone transaction management system*. Several earlier solutions suggest the development of a stand-alone transaction management system. Examples of these are TransCoop (de By et al. 1998) and TSME (Georgakopoulos et al. 1996). To develop a complete TP-monitor with advanced transactional support would require the implementation of every thing from scratch. Although this has several benefits such as removing constraints imposed by pre-existing design decisions, and allowing one to concentrate entirely on new ideas about the extended transaction requirements for the development of a new advanced transaction management system, one would spend too much time building basic transactional support and re-inventing wheels, and innovation would be limited. Moreover, the effort needed – in terms of time and money – would go far beyond works such as ours. As a result, we have chosen to rely on developing our framework based on the middleware principle, thus benefiting from its openness and extensibility.

- *Complete implementation of specific ETMs*. This work has not addressed the implementation of specific sets of extended transaction models (ETMs) – e.g., Sagas, Open nested transaction model, Semantic based concurrence control, Cooperative transaction hierarchy (see Section 4.2). Although these issues have been addressed in ASSET (Biliris et al. 1994) and RTF (Barga 1999). On the one hand, this can be regarded as a limitation of CAGISTrans. On the other hand, given the diversity of existing models, combined with the lack of consensus on which models that suit which situations, we have put our emphasis on extracting beneficial features of relevant models, and exploiting their combination to provide efficient, relevant support.

- *Stand-alone specification language*. An alternative to our extensive use of XML would have been the development of a native transaction model specification language. In fact, at an early stage of this work, we investigated the use of LOTOS (van Eijk et al. 1989) as a basis for a specification. The advantage of this could have been the possibility to exploit the language's formality and comprehensiveness. However, taking users' competence and knowledge into account, we might have ended up with a language which only few could use, due to lack of user-friendliness. Rather, we have stressed the use of XML to enable portability and interoperability. Moreover, due to its extensive use along with the Web,

we may anticipate that most users are familiar with the basic ideas of XML.

- *Full recovery mechanisms*. A possible shortcoming of our CAGISTrans system is its rather incomplete recovery support. At the time we designed our system, this was beyond the scope of our work. However, our CAGISTrans system does allow the use of underlying database management systems to do such things as logging for recovery purposes. A problem first arises when the underlying resource bases are other than legacy databases, which calls for CAGISTrans recovery management. Our system provides a simple logging mechanism to manage transaction aborts.

- *Study on how people work*. The CSCW literature – cf. (Schmidt and Bannon 1992, Grudin 1994) – argues the necessity of studying how people work prior to development of system support, thus discovering sociological and psychological factors that affect the way they carry out their work and use computers. However, as we initially pointed out, although the chances of success might increase such as in terms of acceptability – this is beyond the scope of our work. Providing support covering all aspects of cooperative work is neither possible nor feasible. Therefore, we have rather attempted to develop a framework that should cover most user needs.

11.3. Consolidated comparison with other work

This section compares our CAGISTrans framework with other relevant work.

Our CAGISTrans framework differs from other relevant work in our combination of support for explicit customisation of non-ACID requirements, user-defined correctness criteria, explicit support for applied policies, dynamic restructuring and workspace support. This is summarised in Table 11.2. As illustrated some features have been adopted from existing transaction models and frameworks, and then extended in forming the CAGISTrans framework. This is explained in the following.

- *Explicit customisation of non-ACID requirements.* To my knowledge, support for explicit definition of non-ACID requirements to specific applications has not been proposed before. Existing frameworks let non-ACID requirements be implicitly tailored, but do not allow users to define them in accordance with their application's needs.

CAGISTrans features	Inspiration or origin	Supported by other frameworks	Extension or difference
Explicit customisation of non-ACID requirements	None or not known.	Done implicitly in all other relevant frameworks – i.e., explicit specification not supported.	Explicit support.
User-defined correctness criteria	*Conflicts*: Cooperative transaction hierarchy & Semantic-based concurrency control. *Permits*: ASSET. *Demands*: Patterns as in Cooperative transaction hierarchy.	TSME: Through correctness dependencies. ASSET: Through permits.	The combination of all three constraint tools. Dynamic support in terms of runtime modifications.
Explicit support for applied policies	None or not known.	Not supported by any other relevant framework.	Brand new.
Dynamic restructuring	Splitting as in Split and Join transaction model. Delegation as in ACTA.	RTF: Dynamic restructuring. ASSET: Delegation of operations (static).	Support for both lock and operation delegation. Application of user-defined correctness criteria.
Integrated Workspace support	Classical and extended[a] check-in and check-out models, & Coo, EPOS and TransCoop.	Not supported by any other relevant framework.	The combination of unlimited nested structure, flexible workspace operations, & user-defined coordination.

Table 11.2 A summary of CAGISTrans features and their relation to other frameworks.

a. Extended check-in/check-out operations which originated from (Kim et al. 1984, Bancilhon et al. 1985).

- *User-defined correctness criteria*. Patterns and conflicts as user-defined correctness criteria tools were originally proposed by (Skarra 1989). They were improved in the Cooperative Transaction Hierarchy (Nodine and Zdonik 1992), where they were represented as state-machines. The demands and conflicts of our CAGISTrans framework are built on these concepts. However, in contrast to patterns, demands are represented as directed graphs identifying and representing sequences of actions required to ensure correctness. In addition, while a complete set of patterns has to be defined before transaction execution – without any possibility for re-definition during runtime, demands can be modified while transactions are being executed. Moreover, our conflicts are

defined as tabular relationships rather than with a state-machine. In fact, our conflict concept is more similar to that associated with semantic-based concurrency control than those proposed in (Nodine and Zdonik 1992). See (Ramamritham and Chrysanthis 1997) for a discussion of semantic-based concurrency control issues. The benefit here is that when transactions are to be validated, instead of checking a state-machine which is usually complex, CAGISTrans utilises simpler tabular inquiries. Finally, permits were originally proposed with ASSET (Biliris et al. 1994). Our use of the concept also allows flexible sharing. However, permits in CAGISTrans are accompanied by conflicts and demands. For example, we utilise permits both to override specific conflicts and to enable concurrent accesses to locked objects. Hence, CAGISTrans aims at providing a more flexible, but controlled type of sharing.

- *Explicit support for applied policies*. Applied policies in CAGISTrans determine the relevant mechanisms, and specify rules for how and when to use them. Although the distinction between mechanisms and rules has long been known in both the database and the CSCW communities, to my knowledge, it is still not addressed in connection with transaction models and frameworks. Hence, our use of this concept, allowing users to explicitly fit policies in accordance with the needs of the applications seems unique.

- *Dynamic restructuring.* This concept was originally proposed in the Split and Join transaction model (Kaiser and Pu 1992). CAGISTrans applies a similar approach to restructure transactions while they are being executed. The differences lie in the way the restructuring is performed. CAGISTrans realises dynamic restructuring by combining transaction *splitting* – from the Split and Join transaction model – with the notion of *delegation* – which originated with ACTA (Chrysanthis and Ramamritham 1994). Finally, while Split and Join transactions apply serialisability as the correctness criterion, CAGISTrans allows user-defined criteria.

- *Integrated workspace support*. In the context of transactions, the workspace concept has been extensively used in EPOS (Conradi et al. 1995, Conradi et al. 1997), Coo (Godart et al. 1996), and TransCoop (de By et al. 1998, Wäsch 1999), among others. In brief, EPOS and Coo use temporary, shared sub-databases (scratch-pads) for data exchange and integration work. TransCoop focuses on exchange of operations instead of exchanging data between private and public workspaces, while cor-

rectness control is handled through history validation and merging mechanisms. Our workspace concept differs from these in combining several aspects. First, we use a nested workspace structure that applies *unlimited nesting levels* to regulate the degree of sharing. To our knowledge, existing approaches are restricted to *two* – i.e., private and public, and *three levels* – i.e., public, semi-public, and private. Second, our CAGISTrans framework applies several extended workspace operations enhancing workspace interaction – sharing basic ideas of those in (Kim et al. 1984, Bancilhon et al. 1985). Finally, our approach utilises user-defined constraint tools – cf., conflicts, permits, and demands – for coordinated workspace access.

Chapter 12
Conclusions and Future Work

This chapter summarises the major achievements in this thesis and points out some directions for further work.

12.1. Important themes

The diversity of cooperative work highly motivates the possibility to customise all offered support, including transactional support. There are many transaction models and a few transactional frameworks that have provided useful foundations for such support. Still, there are problems that must be faced due to the aforementioned diversity. This thesis has attempted to address the following two main challenges:

- *Dynamic nature of cooperative work*. An important topic that the work in this thesis has focused on, is the dynamic nature of cooperative work. Our objective has been to extract beneficial features of existing models and frameworks and then extend these in forming a new framework that is able to meet this challenge. The fundamental idea in our framework is to distinguish between design time and runtime management of transactions. In this way, predictable parts of a transaction model can be specified before a transaction is executed, while parts that are not possible to reason about a priori can be specified at runtime. Hence, our solution provides transactional support that not only can be customised to suit specific applications, but also refined in accordance with changes in the environment at runtime.

- *Heterogeneous aspect of cooperative environments*. Another important topic that has been focused on, is the heterogeneity aspect of cooperative

environments. In addressing this aspect, we have developed a system exploiting the benefits of the middleware principle. The CAGISTrans architecture allows the use of diverse resource management systems and diverse types of applications.

12.2. Contributions of this thesis

A major contribution of this thesis has been the development of the CAGISTrans framework providing adaptable transactional support for cooperative work. Although some of the approaches applied in our work are not new, we believe that the way we integrate these techniques and extend them to offer such adaptable support, is unique. This uniqueness includes our combination of *explicit customisation of non-ACID-requirements*, *user-defined correctness criteria, explicit support for applied policies, dynamic restructuring*, and *integrated workspace support*. The resulting CAGISTrans framework is able to support the basic features of dynamic, heterogeneous transaction management, allowing users to specify models and letting the system execute their transactions in a flexible, but controlled manner.

The main achievements of this work can be summarised as follows:

- *Customisation and integration*. This thesis has accentuated the importance of providing both a possibility to customise transaction models to specific applications and an integrated workspace management to improve the support for controlled sharing of resources. Through this, our transactional framework has attempted to bridge the gap between the rigidity of traditional transaction processing and the flexibility required by cooperative work.

- *Separation and organisation*. The development of our transactional framework has produced a useful way to organise transaction model elements. A fundamental feature here is the separation between characteristics and execution specifications of transactions. This separation has allowed both design and runtime time specification of transaction models.

- *Specification and management*. Our transactional framework allows transactional behaviour to be specified and managed at runtime – i.e., dynamic re-specification of transactional behaviour. This improves the support for evolution in cooperative work processes. As a result, the need for a complete a priori knowledge of operations to be carried out can be strongly reduced.

- *Dynamics*. Our transactional framework also allows users to define correctness constraints in accordance with the needs of their applications, thus attaining increased flexibility and improved support for cooperation. Moreover, the user's ability to refine correctness constraints at runtime is useful with respect to meeting new requirements while work is in progress.

- *Heterogeneity*. We have designed and implemented a system in accordance with our CAGISTrans framework. This has been based on the middleware principle, running on a variety of resource management systems. Our CAGISTrans system has clearly proven the applicability of our approaches and concepts.

- *XML usage*. We have designed and applied a transactional modelling language exploiting the simplicity, efficiency and extensibility of XML.

Major parts of our work have been published at several international conferences (Ramampiaro and Nygård 1999, Ramampiaro et al. 1999, Ramampiaro et al. 2000, Ramampiaro and Nygård 2001a, Ramampiaro and Nygård 2001b, Ramampiaro and Nygård 2002). This has given us opportunities to discuss our ideas with other researchers in the field, and has resulted in useful feedback.

12.3. Future work

It is beyond the scope of such a work to address all aspects of transactional support for cooperative work. The following are issues left for further studies. They are divided into two categories comprising further extension of the implementation of the CAGISTrans framework and further research.

12.3.1. Extensions of the CAGISTrans implementation

The directions for extending the implementation of the CAGISTrans framework proceed as follows:

- *Implementation of the remaining CAGISTrans components.* Chapters 8 and 9 have discussed the design and implementation of our framework. As indicated there, we were not able to implement all components of the CAGISTrans framework within the limited time boundaries of this work. An important work will thus be to implement the remainder of our framework. This will not only contribute to a better demonstration of the capabilities of CAGISTrans, but also reveal further issues that we may have to address.

- *Improvement of the user interface.* There is a need for further improvement of the graphical user interface (GUI). Currently, we are working on the development of a transaction model specification tool. This mainly focuses on improving the user friendliness of the CAGISTrans specification environment, allowing the user to visually define transaction models, hence avoiding the need for "hard-coding" of models in XML.

12.3.2. Further research

Several issues have been left for further research that include the following topics:

- *Human interface and the CAGISTrans framework.* Although the CAGISTrans approach has attempted to find the trade-off between total system control and user intervention, there is still a need to help users accomplish their task efficiently through a CAGISTrans system. This includes developing proper guidelines for utilising the CAGISTrans framework. Several approaches from the human computer interface (HCI) community may here be used as a starting point. However, it is, as mentioned earlier, also important to investigate the proper trade-off between *supporting* cooperation and *controlling* cooperation, and how this can be realised in a transactional framework like CAGISTrans. The result from such an investigation would, for instance, be useful in terms of providing relevant cooperation mechanisms.

- *Support for versioning and merging mechanisms.* The use of versions has been advocated as an important means to handle concurrency in design environments. See for example the discussion in (Korth and Speegle 1994). Also, as pointed out by Sommerville (2001), versioning is necessary to maintain different states for different design objects. To further improve the applicability of our CAGISTrans – e.g., in design environments, an investigation of how to incorporate this is necessary. With this again follows a need to develop support for merging operations to facilitate the management of objects that exist in several different versions.

- *Further evaluation and porting to "real world" applications.* Ideally, all research work should be evaluated in the "real world". A continuation of our CAGIS project – CAGIS-II – is currently being conducted, evaluating the different components of CAGIS with respect to collaborative learning. Another continuation is the MOWAHS-project – MObile Work Across Heterogeneous Systems (Nygård et al. 2000) – that the

present author is a member of. It will focus on porting our approaches and concepts into mobile and heterogeneous environments. This may reveal additional issues that we have not been able to address in the current work. In conclusion, the above two projects will give us opportunities to further assess the practicality of CAGISTrans in more realistic environments and reveal issues that must be further considered.

Bibliography

(Agrawal et al. 1995): Agrawal, D., Bruno, J. L., Abbadi, A. E., and Krishnaswamy, V. (1995). Managing concurrent activities in collaborative environments. In *Proceedings of the 1st IFCIS International Conference on Cooperative Information Systems (CoopIS'95)*, pages 112–124.

(Badrinath and Ramamritham 1992): Badrinath, B. R. and Ramamritham, K. (1992). Semantics-based concurrency control: Beyond commutativity. *Transactions on Distributed Systems*, 17(1):163–199.

(Bancilhon et al. 1985) Bancilhon, F., Kim, W., and Korth, H. F. (1985). A model of CAD transactions. In Pirotte, A. and Vassiliou, Y., editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 25–33. Morgan Kaufmann.

(Barga 1999): Barga, R. (1999). *A Reflective Framework for Implementing Extended Transactions*. Ph.d. dissertation, Oregon Graduate Institute of Science and Technology.

(Barga and Pu 1997): Barga, R. and Pu, C. (1997). A reflective framework for implementing extended transactions. In Jajodia, S. and Kerschberg, L., editors, *(Jajodia and Kerschberg 1997)*, Chapter 3, pages 63–90. Kluwer Academic Publisher.

(Barghouti and Kaiser 1991): Barghouti, N. and Kaiser, G. (1991). Concurrency control in advanced database applications. *ACM Computing Survey*, 23(3):269–317.

(Batory and Kim 1985) Batory, D. S. and Kim, W. (1985). Modeling concepts for VLSI CAD objects. *TODS*, 10(3):322–346.

(Bernstein 1996): Bernstein, P. A. (1996). Middleware: A model for distributed system services. *Communication of the ACM*, 39(2):86–98.

(Bernstein et al. 1987): Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

(Biliris et al. 1994): Biliris, A., Dar, S., Gehani, N. H., Jagadish, H. V., and Ramamritham, K. (1994). ASSET: A system for supporting extended transactions. In Snodgrass, R. T. and Winslett, M., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 94)*. ACM Press.

(Borghoff and Schlichter 2000): Borghoff, U. M. and Schlichter, J. H. (2000). *Computer Supported Cooperative Work – Introduction to Distributed Applications*. Springer.

(Bradshaw 1997): Bradshaw, J. (1997). *Software Agents*. AAAI Press/The MIT Press.

(Brasethvik and Gulla 1999): Brasethvik, T. and Gulla, J. A. (1999). Semantically accessing documents using conceptual model descriptions. In Chen, P., Embley, D., and Little, S., editors, *Proc of workshop on Web and Conceptual Modelling*, Paris.

(Brasethvik and Gulla 2000): Brasethvik, T. and Gulla, J. A. (2000). Natural language analysis for semantic document modelling. In Metais, E., editor, *Proceedings on 5th International Conference on Application of Nature Language to Information Systems (NLDB'2000)*, Versailles.

(Bray et al. 1998): Bray, T., Paoli, J., and Sperberg-McQueen, C. M. (1998). *Extensible Markup Language (XML) 1.0 - W3C Recommendation 10-February-1998*. W3C, http://www.w3.org/TR/REC-xml.

(Breitbart et al. 1995): Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. (1995). Transaction management in multidatabase systems. In Kim, W., editor, *Modern Database Systems: The object Model, Interoperability, and Beyond*, pages 573–591. ACM Press and Addison-Wesley.

(Breugst et al. 1998): Breugst, M., Busse, I., Covaci, S., and Magedanz, T. (1998). Grasshopper – a mobile agent platform for IN based service environments. In *Proceedings of IEEE Intelligent Networks Workshop (IN' 98)*, pages 279–290, Bordeaux. IEEE CS Press.

(Brockschmidt 1995): Brockschmidt, K. (1995). *Inside OLE*. Microsoft Press, Redmond, Washington, 2nd edition.

(Chrysanthis and Ramamritham 1990): Chrysanthis, P. K. and Ramamritham, K. (1990). ACTA: A framework for specifying and reasoning about transaction structure and behavior. In Garcia-Molina, H. and Jagadish, H. V., editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD92)*, pages 194–203. ACM Press.

(Chrysanthis and Ramamritham 1992): Chrysanthis, P. K. and Ramamritham, K. (1992). ACTA: The saga continues. In Elmagarmid, A. K., editor, *Database Transacation Models for Advanced Applications*, pages 350–397. Morgan Kaufmann.

(Chrysanthis and Ramamritham 1994): Chrysanthis, P. K. and Ramamritham, K. (1994). Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491.

(Conradi et al. 1995): Conradi, R., Hagaseth, M., and Liu, C. (1995). Planning support for cooperating transactions in EPOS. *Information Systems*, 20(4):317–326.

(Conradi et al. 1996): Conradi, R. et al. (1996). CAGIS – Cooperating Agents in the Global Inforamtion Space. IDI-NTNU Project proposal. Accepted for financing by the Research Council of Norway.

(Conradi et al. 1997): Conradi, R., Larsen, J.-O., Nguyen, M., Wang, A. I., and Liu, C. (1997). Transaction models for software engineering database. In *Proceedings of the Dagstuhl Workshop on Software Engineering Databases*.

(Dayal et al. 1988) Dayal, U., Blaustein, B. T., Buchmann, A. P., Chakravarthy, U. S., Hsu, M., Ledin, R., McCarthy, D. R., Rosenthal, A., Sarin, S. K., Carey, M. J., Livny, M., and Jauhari, R. (1988). The HiPac project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70.

(Dayal et al. 1995) Dayal, U., Hanson, E. N., and Widom, J. (1995). Active database systems. In Kim, W., editor, *Modern Database Systems: The object Model, Interoperability, and Beyond*, pages 434–456. ACM Press and Addison-Wesley.

(Dayal et al. 1990): Dayal, U., Hsu, M., and Ladin, R. (1990). Organizing long-running activities with triggers and transactions. In Garcia-Molina, H. and Jagadish, H. V., editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 204–214. ACM Press.

(de By et al. 1998): de By, R. A., Klas, W., and Veijalainen, J. (1998). *Transaction Management Support for Cooperative Applications*. Kluwer Academic Publ.

(DeSanctis and Gallupe 1987): DeSanctis, G. and Gallupe, R. B. (1987). A foundation for the study of group decision support systems. *Management Science*, 33(5):589–609.

(Ellis et al. 1991): Ellis, C. A., Gibbs, S. J., and Rein, G. L. (1991). Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):9–28.

(Elmagarmid 1992): Elmagarmid, A. K. (1992). *Database Transaction Models for Advanced Applications*. Morgan Kaufman.

(Elmagarmid et al. 1992): Elmagarmid, A. K., Leu, Y., Mullen, J. G., and Bukhres, O. (1992). Introduction to advanced transaction models. In Elmagarmid, A. K., editor, *Database Transacation Models for Advanced Applications*, pages 35–52. Morgan Kaufmann.

(Farshchian 2001): Farshchian, B. (2001). *A Framework for Supporting Shared Interaction in Distributed Product Development Projects*. Dr.ing. thesis, Dept. of Computer and Information Science, Norwegian University of Science and Technology. Trondheim.

(Finin et al. 1994): Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). KQML as an agent communication language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463. ACM Press.

(Fussell et al. 1981) Fussell, D., Kedem, Z. M., and Silberschatz, A. (1981). A. deadlock removal using partial rollback in database systems. In *Proceedings of the ACM SIGMOD International Conference on management of data*, pages pp. 65–73, Ann Harbor, Michigan. ACM Press.

(Garcia-Molina and Salem 1987): Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 87)*, pages 249–259.

(Georgakopoulos et al. 1994): Georgakopoulos, D., Hornick, M. F., Krychniak, P., and Manola, F. (1994). Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the 10th International Conference on Data Engineering (ICDE 94)*, pages 462–473. IEEE Computer Society.

(Georgakopoulos et al. 1996): Georgakopoulos, D., Hornick, M. F., and Manola, F. (1996). Customizing transaction models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):630–649.

(Godart 1993): Godart, C. (1993). COO: A transaction model to support cooperating software developers coordiantion. In *Proceedings of the 4th European Software Engineering Conference, Garmisch, LNCS 717*, pages 361–379.

(Godart et al. 1996): Godart, C., Canals, G., Charoy, F., Molli, P., and Skaf, H. (1996). Designing and implementing COO: Design process, architectural style, lessons learned. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, pages 342–352. IEEE-CS Press.

(Gray 1981): Gray, J. (1981). The transaction concept: Virtues and limitations. In *In Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 81)*, pages 144–154. IEEE Computer Society Press.

(Gray et al. 1981) Gray, J., McJones, P. R., Blasgen, M. W., Lindsay, B. G., Lorie, R. A., Price, T. G., Putzolu, G. R., and Traiger, I. L. (1981). The recovery manager of the system R database manager. *ACM Computing Surveys*, 13(2):223–243.

(Gray and Reuter 1993): Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

(Gray et al. 1975) Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. (1975). Granularity of locks and degrees of consistency in a shared database. In *Proceedings of the 1st International Conference on Very Large Databases (VLDB '75)*, pages 25–33.

(Greenberg and Marwood 1994): Greenberg, S. and Marwood, D. (1994). Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proceedings of the conference on Computer supported cooperative work*, pages 207–217. ACM Press.

(Greenberg and Roseman 1998): Greenberg, S. and Roseman, M. (1998). Using a room metaphor to ease transitions in groupware. Research report 98/611/02, Department of Computer Science, University of Calgary.

(Greif and Sarin 1987): Greif, I. and Sarin, S. (1987). Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2):187–211.

(Grudin 1994): Grudin, J. (1994). CSCW: History and focus. *IEEE Computer*, 27(5):19–26.

(Gutwin et al. 1996): Gutwin, C., Greenberg, S., and Roseman, M. (1996). Supporting awareness of others in groupware. In *Proceedings of the CHI '96 conference companion on Human factors in computing systems: common ground*, page 205. ACM Press.

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

(Harrison et al. 1995): Harrison, C. G., Chess, D. M., and Kershenbaum, A. (1995). Mobile agents: Are the a good idea? Technical report, BM T.J. Watson Research Center.

(Jain et al. 1999): Jain, A. K., Aparico, M., and Singh, M. P. (1999). Agents for process coherence in virtual enterprises. *Communication of the ACM*, 42(3):62–69.

(Jajodia and Kerschberg 1997): Jajodia, S. and Kerschberg, L. (1997). *Advanced Transaction Models and Architectures*. Kluwer Academic Publisher.

(Kaiser 1994): Kaiser, G. E. (1994). Cooperative transactions for multi-user environments. In Kim, W., editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, Chapter 20, pages 409–433. ACM Press.

(Kaiser and Pu 1992): Kaiser, G. E. and Pu, C. (1992). Dynamic restructuring of transactions. In Elmagarmid, A. K., editor, *Database Transacation Models for Advanced Applications*, pages 265–295. Morgan Kaufmann.

(Kim et al. 1984) Kim, W., Lorie, R. A., McNabb, D., and Plouffe, W. (1984). A transaction mechanism for engineering design databases. In Dayal, U., Schlageter, G., and Seng, L. H., editors, *Proceeding of the 10th International Conference on Very Large Data Bases (VLDB '84)*, pages 355–362, Singapore. Morgan Kaufmann.

(Kiniry and Zimmerman 1997): Kiniry, J. and Zimmerman, D. (1997). A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4):21–33.

(Klein 1991): Klein, J. (1991). Advanced rule driven transaction management. In *Proceedings of the 36th IEEE Computer Society International Conference (COMPCON)*, pages 562–567. IEEE, IEEE CS Press.

(Korth 1983): Korth, H. F. (1983). Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79.

(Korth et al. 1990): Korth, H. F., Levy, E., and Silberschatz, A. (1990). A formal approach to recovery by compensating transactions. In McLeod, D., Sacks-Davis, R., and Schek, H.-J., editors, *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB '90)*, pages 95–106. Morgan Kaufmann.

(Korth and Speegle 1994): Korth, H. F. and Speegle, G. D. (1994). Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *TODS*, 19(3):492–535.

(Lange and Oshima 1998): Lange, D. and Oshima, M. (1998). *Programming and Deploying Java[tm] Mobile Agents with Aglets*. Addison Wesley.

(Levine et al. 1992): Levine, J., Mason, T., and Brown, D. (1992). *lex & yacc*. O'Reilly, 2nd edition.

(Levy et al. 1991) Levy, E., Korth, H. F., and Silberschatz, A. (1991). A theory of relaxed atomicity (extended abstract). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 95–109, Montreal. ACM Press.

(Lynch 1983): Lynch, N. A. (1983). Multilevel atomicity - a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502.

(Mariani and Rodden 1996): Mariani, J. A. and Rodden, T. (1996). Cooperative information sharing: Developing a shared object service. *The Computer Journal*, 39(6):455–470.

(Marx 1867): Marx, K. (1867). *Capital : A Critique of Political Economy*, volume 1. Penguin Classics, 1990. English translation by Ben Fowkes, reprint edition.

(Mehrotra et al. 1998): Mehrotra, S., Rastogi, R., Korth, H. F., and Silberschatz, A. (1998). Ensuring consistency in multidatabases by preserving two-level serializability. *TODS*, 23(2):199–230.

(Mehrotra et al. 2001): Mehrotra, S., Rastogi, R., Yuri Breitbart, H. F. K., and Silberschatz, A. (2001). Overcoming heterogeneity and autonomy in multidatabase systems. *Information and Computation*, 167(2):137–172.

(Mohan 1994): Mohan, C. (1994). Tutorial: Advanced transaction models - survey and critique. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 94)*, page 521.

(Mohan et al. 2000) Mohan, C., Barber, R., Watts, S., Somani, A., and Zaharioudakis, M. (2000). Evolution of groupware for business applications: A database perspective on Lotus Domino/Notes. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 684–687. Morgan Kaufmann.

(Mohan and Narang 1994) Mohan, C. and Narang, I. (1994). ARIES/CSA: A method for database recovery in client-server architectures. In Snodgrass, R. T. and Winslett, M., editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 55–66. ACM Press.

(Moss 1982): Moss, J. E. B. (1982). Nested transactions and reliable computing. In *Proceedings of the 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*.

(Nodine and Zdonik 1992): Nodine, M. H. and Zdonik, S. B. (1992). Cooperative transaction hierarchies: Transaction support for design applications. *VLDB Journal*, 1(1):41–80.

(Nygård et al., 2000): Nygård, M. et al. (2000). MOWAHS – mobile work across heterogeneous systems. IDI-NTNU Project proposal. Accepted for financing by the Research Council of Norway.

(Object Management Group 1998): Object Management Group (1998). *The Common Object Request Broker: Architecture and Specification*. OMG, v2.2 edition.

(ObjectSpace Inc. 1997): ObjectSpace Inc. (1997). Voyager core package: Technical overview. Technical White Paper.

(Olle et al. 1982): Olle, T. W., Sol, H., and Verrijn-Stuart, A. A., editors (1982). *Proceeding of the IFIP WG 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies, Noordwijkerhout*. IFIP.

(Özsu and Valduriez 1991): Özsu, M. T. and Valduriez, P. (1991). *Principles of Distributed Database Systems*. Prentice-Hall.

(Papadimitriou 1979): Papadimitriou, C. H. (1979). The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653.

(Papows 1995) Papows, J. (1995). Notes for lotus and the world: Sighting the goal. In Coleman, D. and Khanna, R., editors, *Groupware: Technology and Applications*, Chapter 7, pages 201–223. Prentice Hall, Second edition.

(Phillips and Pugh 1994): Phillips, E. and Pugh, D. (1994). *How to Get a Phd : A Handbook for Students and Their Supervisors*. Open University Press, 2nd edition.

(Ramampiaro et al. 1999): Ramampiaro, H., Divitini, M., and Petersen, S. A. (1999). Agent-based groupware: Challenges for cooperative transaction models. In J. Estublier et. al, editors, *Proceedings of the International Process Technology Workshop (IPTW 99)*. Pages 18-22, Villard de Lens.

(Ramampiaro and Nygård 1999): Ramampiaro, H. and Nygård, M. (1999). Cooperative database system: A constructive review of cooperative transaction models. In Kambayashi, Y. and Takakura, H., editors, *Proceedings of the 1999 International Symposium on Database Application in Non-Traditional Environment (DANTE 99)*, pages 315–324, Kyoto. IEEE Computer Society Press.

(Ramampiaro and Nygård 2001a): Ramampiaro, H. and Nygård, M. (2001a). CAGISTrans: A transactional framework for cooperative work. In Sha, E., editor, *Proceedings of the 14th International Conference on Parallel and Distributed Computing Systems (PDCS 2001)*, pages 43–50, Dallas. ISCA.

(Ramampiaro and Nygård 2001b) Ramampiaro, H. and Nygård, M. (2001b). CAGISTrans: Providing adaptable transactional support for cooperative work. In Altinkemer, K. and Chari, K., editors, *Proceedings of the 6th INFORMS Conference on Information Systems and Technology (CIST 2001)*, pages 3 – 29, Florida. INFORMS.

(Ramampiaro and Nygård 2002) Ramampiaro, H. and Nygård, M. (2002). Supporting customisable transactions for cooperative work: An experience paper. In *Proceedings of the 2002 Western Multi conference (WMC 2002) – Collaborative Technologies Symposium 2002 (CTS 2002)*, San Antonio. (To appear).

(Ramampiaro et al. 2000): Ramampiaro, H., Wang, A. I., and Brasethvik, T. (2000). Supporting distributed cooperative work in CAGIS. Presented at *the 4th IASTED International Conference on Software Engineering and Applications (SEA 2000)*, Las Vegas. Aslo in Hamza, M. H., editor, *Proceedings of the IASTED 2001 International Symposia on Applied Informatics (AI 2001)*, pages 609-616, Innsbruck. IASTED/ACTA Press.

(Ramamritham and Chrysanthis 1997): Ramamritham, K. and Chrysanthis, P. K. (1997). *Advances in Concurrency Control and Transaction Processing: Executive Briefing*. IEEE Computer Society Press.

(Rochkind 1975): Rochkind, M. J. (1975). The source code control system. *IEEE Transaction on Software Engineering*, SE-1(4):364–370.

(Rodden 1991): Rodden, T. (1991). A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353.

(Rodden and Blair 1991): Rodden, T. and Blair, G. (1991). CSCW and distributed systems: The problem of control. In Bannon, L., Robinson, M., and Schmidt, K., editors, *Proceedings of the 2nd European Conference on Computer-Supported Cooperative Work*, pages 49–64.

(Saastamoinen 1995): Saastamoinen, H. T. (1995). *On the handling of exceptions in Information Systems*. PhD thesis, University of Jyvaskyla.

(Schmidt and Bannon 1992): Schmidt, K. and Bannon, L. (1992). Taking CSCW seriously. supporting articulation work. *Computer Supported Work. An International Journal*, 1(1-2):7–40.

(Schmidt and Rodden 1996): Schmidt, K. and Rodden, T. (1996). Putting it all together: Requirements for a CSCW platform. In Shapiro, D., Tauber, M., and Traunmüller, R., editors, *The Design of Computer Supported Cooperative Work and Groupware Systems*, chapter 11, pages 157–175. Elsevier Science B.V.

(Shaw 1995): Shaw, T. W. (1995). Development of an electronic classroom: the promise, the possibilities and the practicalities. *Engineering Science and Education Journal*, 4(2):63 –71.

(Skarra 1989): Skarra, A. (1989). Concurrency control for cooperating transactions in an object–oriented database. *SIGPLAN Notices*, 24(4):466–473.

(Sølvberg 1999): Sølvberg, A. (1999). Data and what they refer to. In Chen, P., Akoka, J., Kangassalo, H., and Thalheim, B., editors, *Conceptual modeling – Current Issues and Future Directions*, LNCS1565, pages 211–226. Springer Verlag.

(Sommerville 2001): Sommerville, I. (2001). *Software Engineering*. Addison Wesley, 6th edition.

(Sommerville and Rodden 1993) Sommerville, I. and Rodden, T. (1993). Environments for cooperative systems development. In *Proceedings of the Software Engineering Environments Conference*, pages 144 –155. IEEE Computer Society Press.

(Spahni et al. 1998): Spahni, S., Scherrer, J.-R., Sauquet, D., and Sottile, P.-A. (1998). Consensual trends for optimizing the constitution of middleware. *ACM SIGCOMM Computer Communication Review*, 28(5).

(Straßer et al. 1997): Straßer, M., Baumann, J., and Hohl, F. (1997). Mole – a java based mobile agent system. In Mühlhäuser, M., editor, *Special Issues in Object Oriented Programming*, pages 301–308. dpunkt Verlag.

(Tichy 1985): Tichy, W. F. (1985). RCS: A System for Version Control. *Software Practice and Experience*, 15(7):637–654.

(Tjahjono 1996): Tjahjono, D. (1996). *Exploring the effectiveness of formal technical review factors with CSRS, a collaborative software review system*. PhD thesis, Department of Information and Computer Sciences, University of Hawaii.

(Tulloch 1995): Tulloch, S., editor (1995). *The Oxford Dictionary & Thesaurus.* Oxford University Press.

(van Eijk et al. 1989): van Eijk, P. H. J., Vissers, C. A., and Diaz, M., editors (1989). *The formal description technique LOTOS*. Elsevier Science Publishers B.V.

(Wang 1999): Wang, A. I. (1999). Experience paper: Using XML to implement a workflow tool. In *3rd Annual IASTED International Conference Software Engineering and Applications*, Scottsdale, Arizona.

(Wang 2000a): Wang, A. I. (2000a). Support for Mobile Software Processes in CAGIS. In Conradi, R., editor, *Seventh European Workshop on Software Process Technology*, Kaprun near Salzburg.

(Wang 2000b): Wang, A. I. (2000b). Using Software Agents to Support Evolution of Distributed Workflow Models. In *Proceedings of International ICSC Symposium on Interactive and Collaborative Computing (ICC'2000)*, page 7pp.

(Wang 2001): Wang, A. I. (2001). *Using a Mobile, Agent-based Environment to support Cooperative Software Processes*. Dr.ing. thesis, Norwegian University of Science and Technology, Dept. of Computer and Information Science, NTNU, Trondheim.

(Wang et al. 1998): Wang, A. I., Larsen, J.-O., Conradi, R., and Munch, B. (1998). Improving cooperation support in the EPOS CM system. In Gruhn, V., editor, *Proceedings EWSPT'98, Weybridge (London), 18-19. Sept. 1998, Springer LNCS 1487*, pages 75–91.

(Wäsch 1999): Wäsch, J. (1999). *Transactional Support for Cooperative Applications*. Phd thesis, GMD/IPSI and Darmstadt University of Technology.

(Weihl 1988): Weihl, W. E. (1988). Commutativity-based concurrency control for abstract data types. *IEEE Transaction on Computers*, 37(12):205 – 214.

(Weikum and Schek 1992): Weikum, G. and Schek, H.-J. (1992). Concepts and applications of multilevel transactions and open nested transaction. In Elmagarmid, A. K., editor, *Database Transacation Models for Advanced Applications*, pages 350–397. Morgan Kaufmann.

(Wong et al. 1999): Wong, D., Paciorek, N., and Moore, D. (1999). Java-based mobile agents. *Communications of the ACM*, 42(3):92–102.

(Wooldridge and Jennings 1998): Wooldridge, M. J. and Jennings, N. R. (1998). Applications of intelligent agents. In Jennings, N. R. and Wooldridge, M. J., editors, *Agent Technology Foundations, Applications, and Markets*, chapter 1. Springer-Verlag.

(Zhang et al. 1999): Zhang, Y., Kambayashi, Y., Kambayashi, Y., Yang, Y., and Sun, C. (1999). On interactions between co-existing traditional and cooperative transactions. *International Journal of Cooperative Information Systems*, 8(1):1–24.

Index

D

R

S

T

U