



GMD Research Series

GMD –
Forschungszentrum
Informationstechnik
GmbH

Jürgen Wäsch

Transactional Support for Cooperative Applications

© GMD 1999

GMD – Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
<http://www.gmd.de>

In der Reihe GMD Research Series werden Forschungs- und
Entwicklungsergebnisse aus der GMD zum wissenschaftlichen, nicht-
kommerziellen Gebrauch veröffentlicht. Jegliche Inhaltsänderung des
Dokuments sowie die entgeltliche Weitergabe sind verboten.

The purpose of the GMD Research Series is the dissemination
of research work for scientific non-commercial use.

The commercial distribution of this document is prohibited,
as is any modification of its content.

Anschrift des Verfassers/Address of the author:

Jürgen Wäsch
Institut für Integrierte Publikations- und Informationssysteme
GMD – Forschungszentrum Informationstechnik GmbH
Dolivostraße 15
D-64293 Darmstadt
E-mail: Juergen.Waesch@gmd.de

D17

Die Deutsche Bibliothek - CIP-Einheitsaufnahme:

Wäsch, Jürgen:

Transactional Support for Cooperative Applications / Jürgen Wäsch.
GMD – Forschungszentrum Informationstechnik GmbH. - Sankt Augustin :
GMD – Forschungszentrum Informationstechnik, 1999
(GMD Research Series ; 1999, No. 12)
Zugl.: Darmstadt, Techn. Univ., Diss., 1999
ISBN 3-88457-361-6

ISSN 1435-2699

ISBN 3-88457-361-6

Preface

The work described in this dissertation was performed at the Integrated Publication and Information Systems Institute (IPSI) of the German National Research Center for Information Technology (GMD), located in Darmstadt, Germany. From April 1994 to June 1997 the research was conducted in the context of the European research project TRANSCOOP (Transaction Management Support for Cooperative Applications). The project was a joint effort together with the University of Twente (The Netherlands) and VTT Information Technology (Finland) and was partially funded by the Commission of the European Communities under the Long Term Research grant No. LTR P8012.

Performing this research and writing the dissertation turned out to be a larger effort than I imagined at the beginning. I could not have completed this work without the support of many people and I am delighted to be able to acknowledge them here.

First of all, I would like to express my gratitude to my advisors Professor Erich J. Neuhold (GMD-IPSI / Darmstadt University of Technology) and Professor Marek Rusinkiewicz (Microelectronics and Computer Technology Corporation / University of Houston). They have been extremely helpful in writing this dissertation and provided me with important feedback and advice. Without their support this thesis would not have been possible.

The initial ideas that lead to the development of the cooperative transaction model CO-ACT were generated together with Professor Marek Rusinkiewicz, who was visiting scientist at GMD-IPSI in Summer 1994, Thomas Tesch, Dr. Achim Kraiß, Professor Wolfgang Klas, and Dr. Peter Muth in a series of brainstorming sessions. These ideas were further elaborated by the TRANSCOOP team in Darmstadt, resulting in a series of publications, most of them co-authored by my colleagues Justus Klingemann and Thomas Tesch. Very special thanks to Justus and Thomas for all the inspiring discussions about the topic and good collaboration. Working with them was a very pleasant experience. I wish them all the best for their own “dissertation projects”. I also would like to thank the other TRANSCOOP members at GMD-IPSI, Dr. Karl Aberer, Professor Wolfgang Klas, and Professor Erich J. Neuhold who were mainly responsible for the management of the project, but also provided constructive criticism and valuable comments on our work.

Furthermore, I wish to thank all the people in Enschede and Helsinki who contributed to the success of the TRANSCOOP project. In particular, I would like to thank Professor Peter Apers, Dr. Rolf de By, Dr. Susan Even, Frans Faase, Arno Lehtola, Olli Pihlajamaa, Professor

Henry Tirri, and Professor Jari Veijalainen for their collaboration and their valuable input and comments. The results of our joint effort can be found in the TRANSCOOP book. Special thanks to our Finnish project partners for organizing meetings in various saunas and near the polar circle at $\pm 25^\circ$ Celsius.

The implementation of the COACT cooperative transaction model within the TRANSCOOP demonstrator system would not have been possible without the help of other people. Ajit Bapat and Michael Löhr helped to re-implement the SEPIA hypermedia system on top of the object-oriented database management system VODAK. My students Tina Gran and Björn Harrer contributed to the implementation of the Java API to VODAK. Daniel Tietze implemented the first version of the Java graphical user interface for SEPIA. Timo Salzrieder graduated about the design and implementation of the initial version of the cooperative transaction manager. My colleague Thomas Tesch implemented most of the final cooperative transaction manager module. Mohsen Sarabadani did his diploma thesis about the design and implementation of the graphical user interface to the TRANSCOOP demonstrator. The TRANSCOOP cooperation manager was implemented by Olli Pihlajamaa from VTT. Furthermore, Stephan Karrer designed and implemented a cooperative calendar manager based on the COACT model within his diploma thesis.

Dr. Karl Aberer deserves special thanks for proofreading the first complete draft of my dissertation and providing valuable input how to improve the thesis. I am very grateful to my advisor Professor Erich J. Neuhold for carefully reviewing the subsequent versions of my thesis. I also wish to thank Susanne Boll, Dr. Peter Fankhauser, Silvia Hollfelder, and Thomas Tesch for providing constructive criticism on several versions of this dissertation. Thanks goes also to Ute Sotnik for proofreading the final version.

I warmly thank my colleagues in the Open Adaptive Information Systems division at our institute and all the other people who worked together with me in several research and industrial projects. They contributed to make my time at GMD-IPSI very interesting.

Some friends made my years in Darmstadt very special and amazing. In particular, I would like to thank Susanne, Achim, Alex, Bertin, Christina, Frank, Georg, Guido, Holli, Ingo, Lars, Oliver, Peter, Peter, Steffi, Tobias, Thomas, but also all the others not mentioned here, for their friendship and for all of the good times with lots of fun.

Finally, I wish to express special gratitude to my parents, Elsbeth and Wilhelm. They instilled into me the desire to learn and the drive to pursue this work. Without their support and patience I never could have come this far.

Jürgen Wäsch
Darmstadt, July 1999

Abstract

Keywords: Cooperative Transaction Model, Computer Supported Cooperative Work, Data Consistency, Database Management Systems, Operations Semantics, Commutativity Relations, History Merging, Data Management and Cooperation in Mobile Computing

The rapid growth of the global information infrastructure opens new opportunities to perform joint work in areas such as cooperative authoring, distributed software development, and business workflows. Cooperative work on shared data requires computer support to coordinate the work of multiple users and to ensure the consistency of data processed within a cooperative application.

Maintaining data consistency in multi-user systems is the classical problem of transaction management. The traditional transaction model addresses environments in which a large number of relatively short-lived transactions access a shared database. It is based on the fundamental properties of atomicity, consistency, isolation, and durability of transactions. However, this transaction model does not meet the requirements of cooperative applications. Isolation of transactions, as guaranteed by the ACID properties, contradicts the need to cooperate. Atomicity of transactions is too strict for interactive environments, where activities are of long duration.

A cooperative transaction model needs to support the interactive execution of long-running activities in which competition for resources, on which the locking protocols are based, is replaced by the need to cooperate. The emphasis, therefore, is not on preventing access to resources, but rather on the semantically correct exchange of information among concurrent activities of cooperating users.

In this dissertation, we present a semantics-based cooperative transaction model, called COACT, that addresses the key features of cooperative applications, i.e., interactive user control, long-duration activities, and multi-user cooperation on shared persistent data. The concepts presented in this thesis are preceded by a thorough analysis of requirements derived from a broad spectrum of cooperative applications, ranging from cooperative hypermedia authoring to workflow applications.

The COACT cooperative transaction model builds on the observation that cooperative work is characterized by alternating periods of individual and joint work. Therefore, COACT combines a workspace concept with a new synchronization mechanism, called history merging. To

allow for individual work, we assign a private workspace to every user participating in a cooperative activity. Additionally, there is a common workspace accessible by all participants. To enable joint work, COACT provides a set of information exchange primitives that are all based on history merging. These primitives allow group members to exchange information among workspaces in order to combine their work into a coherent whole.

To that end, we take an operation-oriented view in COACT. State transitions on workspaces are modeled as operations and are recorded in workspace histories. Information exchange among workspaces is realized by the exchange of operations among workspaces, instead of data objects. The consistency of common work results is determined on the basis of the semantics of operations performed in the workspaces. COACT utilizes both backward and forward commutativity relations for this. The flexibility of our approach is mainly achieved by its consideration of operations semantics for dynamically determining consistent units of work, for detecting and resolving conflicts during a merge, and for selective compensation of operations in workspaces.

Merging histories with pre-specified commutativity relations provides transactional guarantees to a cooperative activity. In particular, we guarantee in COACT that all workspace histories remain legal. Using legality as correctness criterion, history merging relaxes the isolation and atomicity properties of the traditional transaction model, while ensuring consistency of the workspaces.

History merging also provides a basis for management of consistency between disconnected or mobile users who operate independently and, yet, must occasionally reconcile their work with each other or with the stationary system. We show in this thesis how to adapt the COACT cooperative transaction model to the special characteristics of mobile environments in order to support disconnected operation mode.

Besides the conceptual development of a formal transaction model, a proof-of-concept demonstrator has been implemented. We describe in this dissertation how the COACT cooperative transaction model has been implemented using an object-oriented database management system and how the new cooperation facilities can be integrated into a hypermedia authoring system.

We believe that the integration of a cooperative transaction model like COACT with advanced database technology can ease the development and deployment of cooperative applications. Furthermore, history merging provides directions for solving open research problems, like semi-automatic merging of versions and re-integration support in mobile environments and asynchronous replication.

Zusammenfassung

Schlüsselworte: Kooperatives Transaktionsmodell, Computerunterstützte Gruppenarbeit, Datenkonsistenz, Datenbankverwaltungssysteme, Semantik von Operationen, Kommutativitätsrelationen, History Merging, Datenverwaltung und Kooperation in mobilen Umgebungen

Eine Vielzahl von Tätigkeiten, die in heutigen Unternehmen anfallen, werden nicht mehr von Einzelpersonen bearbeitet, sondern gemeinsam in der Gruppe gelöst. Beispiele sind das gemeinsame Erstellen von Dokumenten, Designprozesse, Softwareentwicklung und unternehmensweite Geschäftsprozesse. Allerdings werden kooperative Abläufe von heutigen Anwendungsprogrammen nicht bzw. nur unzureichend unterstützt, so daß die Anwender selbst für einen korrekten Arbeitsablauf und die Konsistenz gemeinsam benutzter Dokument- und Informationsbestände verantwortlich sind.

Die Erhaltung der Konsistenz von Daten bei gleichzeitigem Zugriff mehrerer Benutzer ist die klassische Aufgabe des Transaktionsmanagements in Datenbankverwaltungssystemen. Das traditionelle Transaktionsmodell basiert dabei auf der Annahme einer hohen Zahl relativ kurzer Transaktionen, die atomar und isoliert voneinander ablaufen.

Kooperative Anwendungen haben allerdings stark abweichende Anforderungen an Transaktionsunterstützung. Ein kooperatives Transaktionsmodell muß strukturierte Arbeitsprozesse von langer Dauer unterstützen und dabei eine flexible, interaktive und konsistenzerhaltende Kooperation der Gruppenmitglieder auf gemeinsam genutzten Datenbeständen ermöglichen.

In dieser Dissertation beschreiben wir ein neuartiges Transaktionsmodell (COACT), das Kooperation zwischen mehreren Benutzern ermöglicht, gleichzeitig aber transaktionsartige Ausführungsgarantien sowohl für die Arbeit der Gruppe als auch für die Arbeit des einzelnen Benutzers gibt. Ausgangspunkt der Dissertation ist eine umfassende Anforderungsanalyse verschiedener kooperativer Anwendungen. Das Spektrum reicht dabei von routinemäßiger Vorgangsbearbeitung bis zu unstrukturierten, kreativen Aufgaben, die stark interaktiv und kooperativ durchgeführt werden.

Alle untersuchten Anwendungen sind durch abwechselnde Phasen individueller und gemeinsamer Arbeit gekennzeichnet. Um unabhängige Arbeit zu ermöglichen, wird im COACT Modell jedem Benutzer ein privater Arbeitsbereich zugeordnet. Zusätzlich existiert ein gemeinsamer Arbeitsbereich für die gesamte kooperative Aktivität, der allen Kooperationsteilnehmern zugänglich ist. Die Kooperation der Benutzer untereinander, d.h. der kontrollierte Austausch

von Informationen zwischen den verschiedenen Arbeitsbereichen, wird durch spezielle Austauschprimitive ermöglicht. Zentraler Aspekt dabei ist die nahtlose Integration der Arbeitsergebnisse einzelner Benutzer in einem Arbeitsbereich. Dazu wird ein neuer leistungsfähiger Integrationsmechanismus entwickelt, das sogenannte History Merging Verfahren. Dieses ist in der Lage scheinbar bestehende Integrationskonflikte auf einer semantisch höheren Ebene aufzulösen.

Hierbei verfolgt COACT einen operationsbasierten Ansatz. Alle in einem Arbeitsbereich ausgeführten Operationen werden in einer Historie des Arbeitsbereichs protokolliert. Der Austausch von Informationen zwischen Arbeitsbereichen erfolgt nicht durch den Austausch von Datenobjekten, sondern durch die erneute Ausführung ausgewählter Operationen im Zielarbeitsbereich. Die Datenkonsistenz wird dabei durch das History Merging Verfahren unter Ausnutzung der Semantik von Operationen sichergestellt. Dazu verwenden wir sowohl Vorwärts- als auch Rückwärtskommutativität von Operationen in den Historien. Die Flexibilität unseres Ansatzes ergibt sich dabei aus der Berücksichtigung von Kommutativitätsrelationen zur dynamischen Bestimmung konsistenter Arbeitseinheiten, zur Erkennung und Auflösung von relevanten Konflikten bei der Integration von Arbeitsergebnissen und zur selektiven Kompensation von Operationen in Arbeitsbereichen.

Formal betrachtet garantiert das History Merging Verfahren die Legalität von Operationen in den Historien der einzelnen Arbeitsbereiche. Eine Operation ist legal, falls ihr beobachtbares Verhalten ihrer funktionalen Spezifikation genügt. Das Korrektheitskriterium der Legalität garantiert die Konsistenz der Daten in den einzelnen Arbeitsbereichen, während die Isolations- und Atomaritätseigenschaft des traditionellen Transaktionsmodells gezielt abgeschwächt werden. Somit wird Kooperation zwischen parallel arbeitenden Kooperationspartnern ermöglicht.

Neben der Unterstützung kooperativer Anwendungen ist das COACT Modell auch zur Datenverwaltung und Kooperationsunterstützung in mobilen Umgebungen geeignet. Die privaten Arbeitsbereiche ermöglichen es mobilen Benutzern, trotz Verbindungsunterbrechung zum stationären System weiterzuarbeiten. Zur korrekten Integration der während dieser Zeit erstellten Arbeitsergebnisse können wiederum auf dem History Merging Verfahren basierende Synchronisationsprimitive verwendet werden.

Um die Anwendbarkeit der entwickelten Konzepte zu demonstrieren wurde das COACT Modell als Erweiterung des objektorientierten Datenbanksystems VODAK implementiert. Das auf VODAK aufbauende Hypermedia-Autorensystem SEPIA wurde um die von COACT angebotenen Kooperationsmechanismen erweitert und mit zusätzlichen Koordinationsmechanismen integriert.

Wir glauben, daß die Integration des kooperativen Transaktionsmodells COACT mit moderner Datenbanktechnologie die Konzeption und Realisierung kooperativer Anwendungen erheblich vereinfacht. Dabei reicht das Anwendungsgebiet der in dieser Arbeit entwickelten Konzepte über die untersuchten kooperativen Anwendungen hinaus. Mögliche Einsatzbereiche sind etwa die teilautomatisierte Kombination von Versionen in Repositorysystemen und die semantische Re-integration von Replikaten in mobilen und asynchron replizierten Umgebungen.

Contents

Preface	iii
Abstract	v
Zusammenfassung	vii
1 Introduction	1
1.1 Transactions in Database Management Systems	2
1.2 Transactions in Cooperative Applications	5
1.3 COACT and the TRANSCOOP Project	6
1.3.1 The CoCoA Coordination Model	8
1.3.2 The COACT Cooperative Transaction Model	8
1.4 Contributions of the Dissertation	9
1.5 Outline of the Dissertation	11
2 Requirements Analysis and Related Work	13
2.1 Analysis of Selected Classes of Cooperative Applications	14
2.1.1 Cooperative Hypermedia Document Authoring	14
2.1.2 Design for Manufacturing	16
2.1.3 Multi-User Software Development	17
2.1.4 Workflow Applications	18
2.1.5 Common Characteristics and Differences	19
2.2 Requirements for a Cooperative Transaction Model	21
2.2.1 Illustrating Example	21
2.2.2 Consolidated Requirements	22
2.2.2.1 Relaxed Atomicity	22
2.2.2.2 Support for Retraction of Decisions	22
2.2.2.3 Support for Consistent Exchange of Work	23
2.2.2.4 Support for Private and Shared Data	24
2.2.2.5 Coordination of Individual and Joint Work	24
2.2.2.6 Requirements Outside of Cooperative Transaction Models	25
2.3 Transaction Support for Cooperative Applications: An Overview	25
2.3.1 Semantics-Based Concurrency Control for Abstract Data Types	26

2.3.2	Multi-Level and Open-Nested Transactions	27
2.3.3	Dynamic Restructuring of Transactions	29
2.3.4	Check-Out/Check-In, Versioning, and Workspaces	31
2.3.5	Group-Oriented Transaction Approaches	33
2.3.6	Transactional Workflow Models	34
2.3.7	Comparison of Advanced Transaction Models	36
2.3.8	Other Approaches to Support Cooperation	36
2.3.8.1	Groupware Systems	36
2.3.8.2	Workflow Management Systems	38
3	The Cooperative Activity Model	41
3.1	Overview of the Cooperative Activity Model	42
3.2	Model of Computation	46
3.2.1	Data Model	47
3.2.1.1	Objects and Workspaces	47
3.2.1.2	Operation Types	49
3.2.2	Operations	51
3.2.3	Histories	54
3.3	Commutativity	59
3.3.1	Commutativity Relations for Intra-Workspace Dependencies	62
3.3.1.1	Backward Commutativity	62
3.3.1.2	Excursus: Serializability in Update-In-Place Models	66
3.3.1.3	Right Backward Commutativity	68
3.3.2	Commutativity for Inter-Workspace Dependencies	72
3.3.2.1	Forward Commutativity	73
3.3.2.2	Left Forward Commutativity	77
3.3.3	Commutativity Relations Independent of the Update-Model	79
3.4	Cooperative Activities	83
3.4.1	Cooperative Activity Types	83
3.4.1.1	Invocable Operations Types	84
3.4.1.2	Compensation Function	84
3.4.1.3	Merging Relations	85
3.4.2	Sample Cooperative Activity Type for SEPIA	86
3.4.3	Cooperative Activity Instances	90
3.5	History Merging	95
3.5.1	Consistent Units of Work	96
3.5.2	Correct Merging of Histories	100
3.5.3	Compensation	110
3.5.4	COACT Merge Algorithm	117
3.5.4.1	Basic Merge Algorithm	117
3.5.4.2	Example: COACT at Work	122

3.5.4.3	Optimization and Refinements of the Merge Algorithm . . .	127
3.5.5	Asymmetric Relations in the Context of COACT	134
3.5.5.1	Generalization of History Merging	134
3.5.5.2	Other Relations than Commutativity	137
3.6	Discussion	138
3.6.1	Organizing the Work Process: The COCoA Coordination Model	138
3.6.1.1	Specifying Organizational Aspects of Cooperative Activities	138
3.6.1.2	Constraints on Single Workspaces	142
3.6.1.3	Integration of the COACT Cooperative Transaction Model with COCoA	142
3.6.2	Requirements Revisited	143
3.6.3	Related Transaction Models	144
4	Support of Mobile Users in COACT	149
4.1	Mobile Computing	149
4.1.1	Architecture and Characteristics of Mobile Systems	150
4.1.2	Cooperative Work in Mobile Environments	152
4.2	Mobile CoAct Architecture	153
4.3	Information Exchange in Presence of Mobile Users	156
4.4	Disconnected Cooperative Work in COACT	157
4.4.1	Caching and Prefetching on the Mobile Unit	158
4.4.2	Working in Disconnected Mode	160
4.4.3	Re-Integration of Work	161
4.4.4	Failure-Resilience	162
4.5	Comparison with Related Work	162
5	Prototype Implementation of COACT	165
5.1	Overview of the TRANSCOOP Demonstrator System	165
5.2	The VODAK Object-Oriented DBMS	169
5.2.1	VODAK's Data Model	169
5.2.2	Architecture of the VODAK System	170
5.3	The Demonstrator Application: SEPIA	172
5.3.1	Overview of the SEPIA Hypermedia Authoring System	172
5.3.2	The VODAK-Based SEPIA Backend	173
5.3.3	The SEPIA Frontend	174
5.3.4	Extensions of the VML Database Schema Needed for COACT	176
5.4	Implementation of the COACT Model within the TRANSCOOP System	177
5.4.1	Realization of the Workspace Concept	179
5.4.2	The Status Database	180
5.4.3	Cooperation Manager	180
5.4.4	Cooperative Transaction Manager	181

5.4.5	Extended Graphical User Interface	184
5.5	Comparison with the Original SEPIA System	188
6	Conclusions and Further Work	191
6.1	Achievements of the Dissertation	191
6.2	Other Application Areas of History Merging	192
6.3	Further Research Topics	194
	Bibliography	197
	Appendix	233
A	List of Figures	233
B	List of Tables	235
C	List of Examples	237
D	List of Definitions	239
E	List of Lemmas	241
F	List of Propositions	243
G	List of Theorems	245

“The transaction concept has emerged as the key structuring technique for distributed data and distributed computations. Originally developed and applied to database applications, the transaction model is now being used in new application areas ranging from process control to cooperative work. Not surprisingly, these more sophisticated applications require a refined and generalized transaction model.”

– Jim Gray, 1992 (taken from the foreword to [Elm92]) –

Chapter 1

Introduction

The global information infrastructure—open networks, intranets, as well as mobile networks—opens a growing number of opportunities to perform joint work in locally distributed environments and within virtual organizations. This increasingly requires to support human interaction in cooperative work environments at a computing system level, in addition to conventional means, like phone or electronic mail. Examples of such *cooperative applications* are design applications like cooperative document authoring, CAD, CASE, or Design for Manufacturing, groupware systems like corporate calendar managers or meeting schedulers, business workflow management, and mobile information systems.

Most of these cooperative applications are based on shared databases, maintaining documents, design data, or business data. Usually, database management systems (DBMS) are employed to model and to store this information persistently. Thus, a DBMS should support the interactions of users cooperating with each other. This requires support for information sharing, for long duration activities, and for interactive user control. At the same time, a DBMS has to enforce data consistency and application-specific correctness criteria.

Ensuring consistency of shared and persistent data in multi-user environments is the classical task of transaction management in database research. The concept of transactions has been investigated from the early 1970s [Bjo73, Dav73]. The traditional transaction model is based on the assumption that a large number of relatively short-lived transactions access a shared database (online transaction processing). This has led to the emergence of the so-called ACID transaction model [Gra81, HR83, BHG87] that is based on the fundamental principles of atomicity, consistency, isolation, and durability of transactions. This traditional transaction model uses read/write serializability as a natural correctness criterion for the concurrent execution of transactions [Pap86, BHG87].

With the emergence of new classes of applications, the basic concept of transaction needs to be re-examined, since these applications impose quite different requirements on a transaction model. The main characteristic of design applications is *user interaction*. In online transaction processing, programmers prepare transactions and users submit them for execution, whereas in design applications users provide input while the transaction is running and interactively

control the transaction execution. This leads to transactions that are of *long-duration* from a computer system's perspective, and often in human terms as well [Kor95]. Atomicity of transactions is no longer feasible, due to the economic cost of lost work in case of transaction rollback [Kai95].

The shift from batch-oriented to highly interactive transaction processing has become even more important with the emergence and growing acceptance of workflow, groupware, and mobile computing. Transactions in such environments have to support the interactive processing of long-running activities, where the traditional paradigm of *competition* for resources (on which the lock-based concurrency control protocols are based) is replaced by the need to *cooperate*. The emphasis, therefore, is not on preventing access to resources, but rather on the interoperation, i.e., the *semantically correct sharing and exchange of information*, among concurrent activities of cooperating users [RKT⁺95a, WK96].

The traditional transaction model is not appropriate for cooperative applications, for several reasons. Failure atomicity is too strict for interactive environments with long-duration activities. Isolation of transactions contradicts the need for multi-user cooperation on shared data. Serializability is often incompatible with consistency, when concerted effort is necessary to take the shared data from one semantically consistent state to another [Kai95]. On the other hand, for most cooperative applications, DBMS support is still needed, e.g., to provide controlled access to data shared among cooperating users, and to ensure durability of the data. A cooperative transaction model has to allow multiple users to work cooperatively on the data, but still has to provide transactional execution guarantees for the work of single users as well as for the overall cooperative activity.

The main contribution of this dissertation is the development of a semantics-based cooperative transaction model called *Cooperative Activity Model* [RKT⁺95a, WK96, Wäs96, KTW96b, WAN98, KTWK98, KTW97b, KTW97a] which was developed in the context of the TRANSCOOP project. The Cooperative Activity model (COACT, for short) addresses the key features of cooperative applications, i.e., interactive user control, long duration activities, and multi-user cooperation on persistent data. COACT enables cooperation among geographically distributed or even mobile users, and can serve as an application-independent nucleus to build DBMS-based cooperative application on top of it. COACT relaxes the isolation and atomicity property of transactions in a controlled way and substitutes them by new criteria, more suitable for cooperative activities. At the same time, COACT preserves traditional transactional properties like consistency and durability of work performed within cooperative applications.

1.1 Transactions in Database Management Systems

In the following, we present a brief description of standard models for transaction management in traditional DBMSs. Additional information about this topic can be found in several books on database systems and transaction management, including [Gra78, Pap86, BHG87, Mul89, ÖV91, GR93, RC97].

A *database* is a collection of persistent data objects. A *database management system* coordinates all accesses to the database, i.e., a database can only be manipulated by means of its DBMS [VWP⁺98]. Users interact with the DBMS through special application programs. These transaction programs contain (complex) database operations, together with transactional commands, i.e., *begin*, *commit*, and *abort*, marking transaction boundaries. A *transaction* is an execution of a program segment beginning with a *begin* command and ending with a *commit* command (or *abort*). A transaction (execution) is usually modeled as a sequence of elementary database operations, like read and write accesses to database pages on stable storage [BHG87, VWP⁺98].

A primary goal of traditional DBMSs is to allow several users to concurrently access the underlying shared database. Additionally, a DBMS has to protect its users from hardware and software failures. Therefore, transactions are expected to satisfy the following four conditions, known as the ACID properties [Gra81, HR83].

- **Atomicity** refers to the fact that either all of the database operations of a transaction must be executed, or none of them. Thus, a transaction is treated as an indivisible unit of work.
- **Consistency** requires a single transaction to be correct, i.e., if executed alone, the transaction transfers the database from one consistent state to another consistent state.
- **Isolation** requires each transaction to observe a consistent database state, i.e., a transaction executes without any interference from other concurrent transactions.
- **Durability** requires the results of a committed transaction to be made permanent in the database, surviving any subsequent DBMS failures.

Isolation has also been denoted as *execution atomicity*, and atomicity and durability have been referred to as *failure atomicity* [Mos85, Mos87]. Commonly, consistency is attributed to transaction programs, i.e., it is required that transactions are *programmed* in a way that each preserves the consistency of the database at its boundaries. Hence, the goal of a transaction management mechanism is to guarantee atomicity, isolation, and durability for the interleaved execution of transactions [VWP⁺98]. This is done by combining two different sets of protocols. The first set, known as *concurrency control protocols*, deals with concurrency anomalies [BHG87, GR93] and ensures the isolation property of transactions. The second one, referred to as *recovery protocols*, addresses failure anomalies [BHG87, GR93] and ensures atomicity and durability properties. Note that usually recovery and concurrency control protocols depend on each other.

In traditional transaction models, the isolation property is tackled by the formal notion of *serializability* of transaction executions, i.e., *histories*. The key idea behind serializability is as follows. Assuming that individual transactions are consistent, a serial execution of a set of transactions, i.e., one transaction after the other, must be correct. Since such serial histories are assumed to be correct, any interleaved transaction execution, whose behavior or effect are

the same as some serial history, must also be correct. Such histories are called *serializable*. Usually, serializability is defined by means of conflicting read and write operations to database pages. In general, two database operations *conflict* (do not commute) when their effect is order-dependent. For example, in the read/write model, two write operations conflict, as well as a read and a write. *Conflict serializability* requires that each pair of conflicting operations of different transactions appears in the same order as in a serial history [GLPT75, EGLT76, Gra78, BSW79, Pap79]. Whether an execution is conflict-serializable can be easily determined by testing if the corresponding serialization graph is acyclic [GLPT75, Pap86, BHG87] (cf. Section 3.3.1.2). For a detailed treatment of serializability theory, we refer to [BHG87, Pap86].

Concurrency control protocols can be classified as *pessimistic* or *optimistic*. The principle underlying pessimistic protocols is that before attempting to perform a database operation, a transaction has to ask for permission to do so. In case of *locking-based protocols*, locks are used to represent permissions to perform a database operation. If the transaction manager detects a conflict by examining a conflict table for (elementary) database operations, a lock request is denied. The requesting transaction is blocked until the release of the conflicting lock. Otherwise, the lock is granted, which usually follows the *two-phase locking protocol* [EGLT76]. Since transactions can have cyclic “wait-for-lock” dependencies, lock-based protocols suffer from *deadlocks* [Obe82, Elm86]. Deadlocks have to be resolved by aborting one of the transactions involved, resulting in loss of work [FKS81]. Another drawback are long-duration waits for locks, e.g., in case of long-running transactions. Two non-locking pessimistic protocols that avoid deadlocks are *timestamp ordering* [BG81, Ree83] and *serialization graph testing* [Bad79, Cas81].

In optimistic concurrency control protocols [KR81, Här84], transactions are allowed to perform conflicting database operations without being blocked—but when they attempt to commit, transactions are validated in order to ensure serializability. Dependent on the validation mechanism, optimistic protocols can be categorized as *forward validation* or *backward validation* [Här84]. Forward validation ensures that a committing transaction cannot invalidate any active transactions. If there exist conflicts, either the transaction that is about to commit is aborted [HCL90] or all conflicting active transactions are aborted [MN82]. Backward validation ensures that the committing transaction has not been invalidated by database operations of recently committed transactions. If there exists a conflict, the transaction cannot commit and is aborted.

Recovery protocols enforce the atomicity and durability properties of transactions [BHG87, GR93]. Recovery protocols need to address the usage of volatile database buffers and the different strategies in propagating updates to the permanent database. To reduce their complexity, it is often required that histories have properties additional to serializability, such as recoverability, avoidance of cascading aborts, strictness [Had83, Had88], and rigorousness [BGRS91].

Recovery protocols use two basic actions on the permanent database. The *undo action* is required for atomicity and undoes updates on the database done by an aborted transaction. The *redo action* is needed for durability and redoes updates on the permanent database done by a

committed transaction. The information necessary for undo and redo actions is usually kept in the *database log* on stable storage. This log keeps track of all operations that modify data objects in the database. In case of a physical log, for each database operation updating a data object, the log records the old value of the object (before-image) and the new value of the object (after-image) [Ver78, HR83]. The before-image is needed by the undo action whereas the after-image is used by the redo action. A logical log contains only the description of the executed high-level database operations and their parameters. This information is used to *re-execute* the database operations and to *compensate*, i.e., undoing, the database operation.

In general, update propagation strategies that allow a transaction to update the permanent database before reaching its commit (steal policy) require undo actions. Protocols that do not update the stable database before transaction commits (no-steal policy) require no undo actions (cf. Section 3.3). If database pages are not necessarily propagated to stable storage during commit processing (no-force policy), redo actions are required to ensure durability of transactions. For a systematic classification of recovery protocols, including checkpointing, we refer to [HR83]. One well-known family of recovery protocols is ARIES [MHL⁺92, RM89, ML92, Moh96]. In a nutshell, ARIES recovers a DBMS after a failure by maintaining *write-ahead* logs [Gra78, MHL⁺92]. Upon a failure, it applies redo log information, i.e., repeats history, of all (committed or uncommitted) transactions since the last checkpoint. Afterwards, it applies the undo log to compensate for the effects of transactions that definitely did not commit prior to the failure.

1.2 Transactions in Cooperative Applications

We argued that requirements originating from cooperative applications are not met by traditional DBMS technology. In particular, the isolation property of traditional transactions contradicts the need to share partial results of transactions while they are still in progress. Atomicity is undesirable, since it may lead to long-duration waits of transactions and results in the roll-back of the complete transaction, e.g., if a failure occurs or transaction validation fails. On the other hand, the need for DBMS technology is obvious, as sharing and exchanging persistent data is the characteristic feature of cooperative applications (cf. Chapter 2).

As a consequence, special solutions have been developed which try to compensate the lack of cooperation support by DBMSs (like in SEPIA [SHH⁺92, SH93, HW92] or the PoliWork system [TV99, TBR98]). Components on top of a DBMS are used which allow the description of a cooperative activity, try to coordinate tasks performed in a cooperative activity, and map them to the basic mechanisms of ACID transactions provided by the DBMS. Experience with such solutions shows that the overall architecture becomes very complicated. The interaction with the DBMS becomes much more complex at the application level [STTB99, TBR98], since the application programmers are explicitly concerned with enforcing data consistency, controlling visibility of data modifications done by cooperating users, and ensuring recoverability of cooperative activities.

From a general perspective of technological progress, the incarnation of such solutions that are always dedicated to a single application is a step backwards, as abstract schemes for different kinds of cooperation are missing. When new applications have to be developed, new systems capturing the cooperative aspects of the applications have to be developed, too [KdBV98].

Therefore, it has been tried to extend the traditional notion of transaction offered by DBMSs in order to fulfill the requirements of cooperative applications. Early work in advancing the traditional transaction model kept (generalized) ACID principles, but added nesting of transactions [Mos82, HR87a, Mos87, BBG89] to provide for fine-grained recovery and intra-transaction parallelism. Other approaches exploited commutativity properties of database operations [Kor83, Wei88a, Her90, BR92] to enhance concurrency.

Later work in the area of *advanced transaction models* addressed especially long-duration activities in design environments, as well as the relaxation of the isolation property to enable cooperation among concurrently running transactions. The gross of work trying to overcome the limitations of the traditional transaction model added semantics to the model and, using this added semantics, was able to avoid some of the drawbacks of the ACID principle [Kor95]. The proposed advanced transaction models differ with regard to the degree they fulfill the requirements for a cooperative transaction model (cf. Section 2.2). Moreover, some of the models lack a formal notion of consistency in case isolation or atomicity properties are relaxed.

In the TRANSCOOP project, our aim was to satisfy the requirements on transactional support for cooperative applications by introducing a set of complementary mechanism, which can be used by an application designer to build cooperative applications, and—at the same time—enforce formal consistency criteria. In the next section, we describe the approach followed in TRANSCOOP and give an overview of the COACT cooperative transaction model described in this thesis.

1.3 COACT and the TRANSCOOP Project

The overall goal of the TRANSCOOP project¹ [TC94, AEF⁺97, dBKV98] was to develop transaction management concepts and mechanisms that are applicable to a broad spectrum of cooperation, ranging from ad-hoc cooperation schemes to well-structured cooperation schemes like workflows. In particular, the objectives of TRANSCOOP were as follows [TC94, KdBV98]:

- to develop a specification language which allows to describe cooperative activities, and which can be supported by a cooperative transaction management mechanism,

¹The ESPRIT-III project TRANSCOOP (Transaction Management Support for Cooperative Applications) was partially funded by the European Commission under the Long Term Research Grant No. ESPRIT LTR P8012. The partners in TRANSCOOP were GMD's Institute for Integrated Publication and Information Systems (Germany), the University of Twente (The Netherlands), and VTT Information Technology (Finland). The project started in April 1994 and was finished in June 1997 with a final Workshop on Extending Data Management for Cooperative Work, held in Darmstadt, Germany [AEF⁺97].

- to implement a toolset on the basis of the specification language in order to allow for simulation and correctness proofs of cooperation schemes,
- to develop a formal cooperative transaction model that was flexible enough to meet the requirements of a broad spectrum of cooperative applications,
- to implement this cooperative transaction model and to integrate it into an existing prototype of an object-oriented DBMS, and
- to implement a demonstrator system to prove the applicability of the TRANSCOOP approach for a selected application scenario.

The research activities in TRANSCOOP started with an analysis of different classes of cooperative applications and case studies (cf. Chapter 2) [TVF⁺95, VTF⁺95, dBKV98]. In particular, we investigated cooperative hypermedia document authoring [TW95c], Design for Manufacturing [VFSE95], and workflow applications [JLP⁺95]. Additionally, we studied existing approaches [KPTV95] and commercial products to support cooperative work [VPL95].

One important result of this requirements analysis was the distinction of two different aspects within cooperative applications [AKT⁺96, VdBA98, WAN98]:

- *Organizational aspects* are mainly concerned with the *coordination* of users involved within a joint effort. Application-dependent constraints are needed to ensure *consistency of the work process*, i.e., the structure and control flow within cooperative activities.
- *Transactional aspects* are mainly concerned with the correctness of sharing and exchange of persistent data among users within a joint effort. Transactional guarantees, i.e., transaction-like properties, are needed to ensure *consistency of the data objects* that are processed within cooperative activities.

This observation is supported by research in the field of CSCW (Computer Supported Cooperative Work) [EGR91, Ell91] that identified a dichotomy between two related concepts: *coordination* and *cooperation*² [EW94b, GAP98]. Coordination is understood as a process by which the individual tasks of the members of a group become better organized by an external entity, in such a way that this work organization leads to a common goal. Cooperation emphasizes the capability of self-organization of those group members, who progress to the final goal through mutual adjustments using data sharing and exchange facilities [GAP98].

As a consequence, we developed two complementary mechanisms in TRANSCOOP. The *coordination model* COCoA addresses mainly organizational aspects, whereas the *cooperative transaction model* COACT deals with transactional aspects of cooperative applications. This thesis is concerned only with the cooperative transaction model.

²Researchers in the CSCW area often use the term “collaboration” instead of “cooperation”. They use “cooperation” as a more general term that comprises coordination, collaboration, and communication [EGR91, Ell91]. Since in the field of database and transaction management the term “cooperation” is commonly used instead of “collaboration”, we use the former term in the remainder of this thesis.

1.3.1 The CoCoA Coordination Model

CoCoA [FEdB96, FEdB98, FEdba97] (cf. Section 3.6.1) offers a rich set of primitives to describe the organizational structure of the work process. A *procedure definition* specifies which operations are *enabled* at each stage of the cooperative activity by means of *steps*. The set of operations that are controlled by the step mechanism includes database operations, data exchange meta-operations (see below), as well as *communications*. The latter are used to initiate step transitions, to terminate steps, and to dynamically instantiate parameterized sub-steps. The step concept and the enabling mechanism of CoCoA are complemented by the possibility of specifying *user roles*. CoCoA is designed on top of the formal specification language LOTOS/TM [EFdB96, EF94, dBEV95], which in turn is based on the ISO standard LOTOS [BB89] and the TM database specification language and design tools, developed at the University of Twente [BBdB⁺96, vKSA⁺95, FvKS94, BdBZ93].

1.3.2 The CoACT Cooperative Transaction Model

CoACT is based on the idea of exchanging partial results between users involved in a *cooperative activity*. In addition to a shared database (*common workspace*), each user has a *private workspace* in which copies of (some of) the data objects reside. A user modifies his private workspace by invoking *data operations*. All data operations are recorded in the *workspace history*, using write-ahead logging. Users can exchange partial results directly with each other or via the common workspace, by means of CoACT's *exchange meta-operations*. After the completion of the cooperative activity, which can be viewed as a long-lived transaction, the result of the cooperative activity is found in the common workspace.

CoACT introduces the *history merging* paradigm to allow users in a cooperative activity to work in parallel and to exchange results, while always maintaining consistency of the workspaces. Note that in CoACT data operations (rather than the data objects themselves) are exchanged among workspaces, resulting in the conceptual “sharing” of operations among workspaces after the merge. The incorporation of partial results into a workspace is realized by *re-executing* operations from a source workspace in the target workspace. In this way the effects of operations exchanged are reflected in the state of the data objects in the target workspace.

History merging exploits the semantics of database operations by means of *commutativity* relations to enforce the correctness of the merge process. Commutativity is used (1) to identify *consistent units of work(spaces)* that can be exchanged and (2) to detect *conflicts* among operations executed in different workspaces. Conflicts that might occur during parallel work are semi-automatically resolved by the CoACT cooperative transaction model: the merge algorithm offers different sets of operations for the merge; one of these operation sets can then be selected. Depending on the selection, conflict resolution may require *undoing* certain operations in the target workspace. This is automatically done by means of *compensation operations*. CoACT's compensation strategy minimizes loss of work during conflict resolution and allows for user-initiated selective undo in private workspaces.

To ease the merge process for the user, the task of selecting an operation set for the merge can be supported by an additional software component without requiring user interaction. In this way, certain merge policies can be specified providing different conflict resolution strategies.

Within the TRANSCOOP project, we have integrated the COACT model into the COCoA specification language. Facilities to define the input for COACT, i.e., *cooperative activity types* consisting of invocable data operations, commutativity relations, and compensation operations, are available in COCoA. Moreover, the invocation of data operations and exchange meta-operations can be constrained using COCoA's enabling mechanism and user roles. COCoA also allows to specify dynamic consistency constraints on workspace histories by means of *execution order rules* (cf. Section 3.6.1).

1.4 Contributions of the Dissertation

This dissertation contributes in various aspects to the state of the art in the area of transactional support for cooperation applications.

We present a thorough analysis of requirements presented by diverse classes of cooperative applications, ranging from cooperative hypermedia authoring, through Design for Manufacturing and multi-user software development, to workflow applications. From this analysis, we derive requirements for our cooperative transaction model. To the best of our knowledge, no requirements analysis done so far comprises such a large spectrum of applications.

Based on the results of our requirements analysis, we developed the COACT cooperative transaction model. The unique feature of COACT is its combination of private workspaces, that allow individual participants to work independently, with a new synchronization mechanism, i.e., history merging, which enable group members to combine their work in order to form a coherent whole. The challenge thereby is to flexibly merge the results of co-workers in such a way that the amount of work that needs to be undone in order to ensure consistency of the workspaces is minimized.

The following example of a group calendar application [Kar98] illustrates this problem. Suppose, a manager maintains a list of open issues in his private workspace. Staff members create meeting entries in their private calendars, which the manager imports into his workspace. The manager then assigns open issues to meetings. These are automatically marked as assigned. If the location of a meeting is changed later on by a staff member, this should be reflected in the manager's workspace when they synchronize their workspaces, but should not result in a conflict, although both users modified the same calendar entry. When the meeting is cancelled, this should result in a conflict with the assignment of open issues. If the manager accepts the cancellation, only that portion of the manager's work that leads to inconsistencies should be discarded. In particular, only the assignment of issues to the cancelled meeting should be undone, but not the open issues themselves.

To provide this flexibility, we take an operation-oriented view in COACT. The sharing of

partial results among workspaces (without final commits) is accomplished by means of sharing and exchanging operations among workspaces. The consistency of shared results is determined based on the semantics of the operations performed on data objects.

To that end, the thesis contains a formal model of computation and a comprehensive discussion of different kinds of commutativity relations, together with important properties of these relations. This includes the novel concepts of “operation views” and an asymmetric variant of forward commutativity.

We introduce a new and original notion of correctness of concurrent executions, based on controlled exchange of operations among workspaces, using the history merging paradigm. Merging histories with pre-specified commutativity rules provides transactional guarantees to a cooperative activity. In particular, we guarantee in COACT that all workspace histories remain *legal*, although information is exchanged. Legality is a novel formal concept beyond conflict serializability and ensures that all data operations in the system behave as expected. This approach relaxes the isolation property of traditional transactions and enables cooperation among users—while always ensuring consistency of their workspaces.

The thesis presents merge algorithms for implementing the information exchange meta-operations of COACT. We utilize backward commutativity [Wei88a, LMWF94] to dynamically identify consistent units of work(spaces) that can be subject to an information exchange. This approach is well-suited for interactively executed “transactions” and relaxes the atomicity property of transactions in a controlled way.

To determine conflicts among operations concurrently executed in different workspaces, we use forward commutativity [Wei88a, LMWF94]. Our concept of identical operations shared among different workspaces allows us to ignore several conflicts during the merge process and, hence, to establish close cooperation.

If the merge cannot be performed without violating the consistency of the target workspaces, the compensation of some operations may be required. We show how backward commutativity together with compensation operations can be used to minimize the loss of work during conflict resolution. Only the portion of the work that leads to inconsistencies needs to be discarded. Additionally, our compensation approach allows users to selectively undo operations in their private workspaces, which is important for the interactive processing of activities.

The design of the COACT model was influenced—to some extent—by other advanced transaction models, in particular, semantics-based concurrency control [Kor83, SS84, Wei88a], open-nested transactions [WS92, MRKN92, MRW⁺93], split/join transactions [PKH88, KP92], and check-out/check-in [Kat90, Kai95]. Like semantics-based concurrency control and open-nested transactions, we use commutativity relations to describe the semantics of operations. Compensation is employed, in a restrictive form, in the open-nested transaction model, too. Split/join transactions support the explicit exchange of operations among ongoing transactions, but is restricted to serializable histories. With respect to the workspace concept, COACT shares some ideas with check-in/check-out models. Rather than exclusively reserving checked-out objects, COACT allows parallel development and flexible integration of work results.

COACT's design is well-suited to support cooperation among geographically distributed users, since it does not rely on central components. CoAct's conceptual model, together with history merging, can also provide the basis for the management of consistency between disconnected and mobile users who work independently and, yet, must occasionally reconcile their work with each other or the stationary system. For this, we introduce an extended system architecture that makes the COACT model also attractive for supporting cooperative work in mobile environments.

Besides the conceptual development of a formal transaction model, the transaction management mechanism has been implemented to practically prove and demonstrate the concepts of COACT. The thesis describes the implementation of the COACT cooperative transaction manager and its integration into the object-oriented DBMS VODAK and the TRANSCOOP demonstrator system. Upon this platform, we have re-implemented the SEPIA hypermedia authoring system as a concept demonstrator—extending it with the new cooperation facilities provided by COACT.

We believe that the integration of a cooperative transaction model like COACT with advanced database technology can ease the development and deployment of cooperative applications. Moreover, history merging is a general synchronization mechanism that can be utilized in various application domains. For instance, history merging provides directions for solving open research problems in related fields like semi-automatic merging in versioning systems, synchronization in realtime groupware systems, transaction support in mobile environments, and asynchronous database replication.

1.5 Outline of the Dissertation

The remainder of this thesis is structured as follows. In Chapter 2, we describe the main features of the investigated cooperative applications. Based on their common characteristics, we present consolidated requirements for a cooperative transaction model. Afterwards, we discuss the state of the art in the area of system support for cooperative applications.

In Chapter 3, we describe the COACT model and its underlying paradigm of cooperation. We present a formal model of computation and introduce the central concept of legality of histories. Based on this, we define various kinds of commutativity relations. Afterwards, we introduce COACT's notion of cooperative activity types and cooperative activities.

The core of the cooperation facilities in COACT is the history merging approach, which is formally defined in Section 3.5. We describe our concept of consistent units of work and the correct merging of histories that permits the semantically correct exchange of information among workspaces. To resolve merge conflicts, we introduce a formal model of selective compensations. In Section 3.5.4, we present merge algorithms that can be used to implement COACT's exchange information exchange facilities, including several refinements and optimizations. Afterwards, we generalize history merging to asymmetric commutativity relations and briefly discuss the applicability of other semantic relations.

We conclude Chapter 3 by describing the CoCoA coordination model, assessing the developed concepts with respect to the identified requirements for a cooperative transaction model, and comparing our approach with the advanced transaction models discussed in Section 2.3.

In Chapter 4, we extend CoACT's system architecture to adapt it to the special characteristics of mobile environments. We discuss new information exchange options and specific problems induced by the possibility of disconnection. We outline how these problems are solved in a mobile CoACT system, thus, enabling “disconnected cooperation” among mobile co-workers.

In Chapter 5, we describe the implementation of the CoACT model within the context of the TRANSCOOP system. In particular, we present the architecture of the TRANSCOOP system, the VODAK object-oriented DBMS and its modeling language, and the VODAK-based implementation of the SEPIA hypermedia authoring system used for demonstration purposes. Afterwards, we describe the implementation of the CoACT transaction manager in more detail. We report on the experience we made during the system design and describe how the new functionality is presented to the users of SEPIA.

The conclusions, including a discussion of potential application areas of history merging and open research issues, are given in Chapter 6.

Chapter 2

Requirements Analysis and Related Work

Ellis et al. [EGR91] denote with the term *cooperative application* any computer system that is used for interactions among users and for sharing and exchanging information among the users. The objective of cooperative applications, also called groupware or CSCW applications, is to support the work of a group of people engaged in a *common task or goal* and to provide an interface to *shared resources* [EGR91, Ell91].

The notion of a common task is crucial to this definition, since it excludes multi-user systems, such as time-sharing or database management systems, whose users do not share a common task. The notion of shared resources is essential, too, since cooperation demands that people working towards a common goal are able to share and exchange artefacts, i.e., documents, forms, software code, product data, etc. [EGR91, Ell91].

Note, it is not required for cooperative applications that the group members be active simultaneously. Cooperative applications that aim at supporting simultaneous activities are called *synchronous*—otherwise, they are called *asynchronous*. Synchronous cooperative applications are highly interactive, real-time applications in which a group of possibly geographically distributed users interacts at the *same* time. Asynchronous cooperative applications do *not* require that all co-workers are present at the same time [Edw97, EMP⁺97, EGR91]. They support an incremental style of working, where each co-worker can work at *arbitrary* times (and arbitrary places) on (parts of) the common task.

This characterization of cooperative applications is quite general. Therefore, the term cooperative application comprises such different kinds of computer systems like email systems, bulletin boards, integrated software development environments, CAD/CAM environments, workflow applications, group decision support systems, Internet telephony, multi-user text editors, desktop video-conferencing, electronic meeting rooms, and many more [EGR91, Rod91, EW94a, BS95, KB95, Bur97]. Since this large spectrum of applications might impose a lot of different and possibly contradicting requirements to underlying computer system support, we start with an analysis of selected classes of cooperative applications. The goal is

to derive requirements for a cooperative transaction model applicable to a broad spectrum of cooperative applications.

The remainder of this section is structured as follows. In Section 2.1, we describe the main characteristics of the application scenarios we have investigated. We classify them with respect to their process structure and summarize their common characteristics. Since the topic of this thesis is to develop a cooperative transaction model, we focus on aspects of data management for cooperative applications in our requirements analysis, rather than on coordination and communication aspects.

In Section 2.2, we present the consolidated requirements for a cooperative transaction model that result from the application analysis. We introduce a simple example from the software development domain to illustrate the requirements we have identified.

In Section 2.3, we discuss related work in the area of transaction support for cooperative applications. In particular, we review some of the most prominent advanced transaction models that address important aspects of data management for cooperative applications. We describe to which degree these models fulfill the requirements developed in Section 2.2. We conclude the chapter by discussing groupware approaches and workflow management systems.

2.1 Analysis of Selected Classes of Cooperative Applications

In the following, we discuss the main characteristics of four typical classes of cooperative applications with respect to requirements for a cooperative transaction model, namely,

- Cooperative hypermedia document authoring,
- Design for Manufacturing,
- Workflow applications, and
- Multi-user software development.

Parts of the analysis of the first three scenarios are taken from the TRANSCOOP project. We, at GMD-IPSI, particularly focused on cooperative hypermedia document authoring. The detailed results of the requirements analysis can be found in [TVF⁺95, VTF⁺95, TVL⁺98], including extensive case studies of real-world cooperative applications.

2.1.1 Cooperative Hypermedia Document Authoring

Hypermedia documents (for short hyperdocuments) form a new class of documents based on the concept of *hypertext* [Nel81, Con87]. A hypertext is a collection of information entities (*nodes*) containing cross-references (*links*), which—with the aid of an interactive browser—allow the reader to navigate within and between the hyperdocuments. If the nodes also contain multimedia information, such as complex pictures and continuous media (e.g., audio and

video), we usually speak of *hypermedia* documents [Con87, MPS91]. The main difference to traditional documents is that hyperdocuments are non-linear. Hyperlinks give authors the ability to connect the information entities in an almost arbitrary way, i.e., a hyperdocument has an associative network-like structure. An important implication of this is the *interactive usage* of hyperdocuments by their readers *and* the authors.

In the following, we describe the cooperative hyperdocument authoring process. Our analysis is based on previous work performed at GMD-IPSI [SHT89, THS90, THH91, HT93, Haa95]. In particular, we have investigated the hypermedia authoring system SEPIA [SHT89, HW92, SHH⁺92] and its underlying cognitive model of cooperative authoring processes [TW95a, TW95c, TW95b, TVL⁺98]. A comprehensive overview of other hypermedia authoring systems can be found in [Haa95].

The Authoring Process in General. Hyperdocument authoring can be seen as a *weakly structured design problem solving process*, mainly characterized by the decomposition of the design problem into smaller subproblems and their solution by interacting activities [HF80, HF86, THS90, HT93, THH95]. An important characteristics of these processes is that the documents to be produced can be described only vaguely in advance. This implies that a particular authoring process usually cannot be specified beforehand.

Despite these observations, it has been tried to identify a generic structure of authoring processes. Hannemann et al. [SHT89, THS90, HT93] decompose the overall design problem into four generic authoring activities, addressing the planning of the process itself (planning problem), the acquisition of background material (content problem), the construction of argumentation chains (argumentation problem), and the construction of a final hyperdocument for the readers (rhetorical problem). It is important to notice that the identified subproblems are highly interdependent.

In the SEPIA authoring system, these subproblems are mapped into different *authoring spaces* with dedicated functionality for their solution, but without constraining the authors in their creativity. With every SEPIA document, a planning space, a content space, an argumentation space, and a rhetorical space are associated [SHT89] (cf. Sections 3.4.2 and 5.3).

Within the different authoring activities, authors *interactively* choose their actions and frequently *undo/redo* them as they go along, i.e., the exact sequence of actions within an activity cannot be determined in advance like in workflow applications. Hypermedia authoring systems have to take this opportunistic nature of the authoring process into account, i.e., they should not restrict the authors to follow a predefined sequence of actions [TW95c].

Hypermedia authoring processes are usually of *long duration*. One reason for this is their opportunistic nature and the lack of a predefined problem solving strategy. Because for authoring processes no optimal solution can be found [Sim81], authors always have the impression that the current solution could be improved. This is the reason that termination predicates for authoring processes can hardly be formalized [TW95c].

The Cooperative Authoring Process. A significant number of documents result from the work of more than one author. In general, *cooperative hypermedia document authoring* (CDA) can be characterized by multiple authors working interactively on a persistent repository of shared hyperdocuments, thereby constructing new hyperdocuments or modifying existing documents in the repository. The essence of cooperation is the exchange of information between the co-authors. CDA involves both the *explicit* exchange of information among certain authors and the *implicit* exchange of documents through the shared repository [TW95a, TW95c, TW95b, TVL⁺98]. Authors need to share and exchange partial results of their activities (e.g., draft documents), while they are still in progress. At the same time, *consistency* of shared documents has to be preserved.

From a theoretical viewpoint, in a cooperative authoring process there appears a new subproblem: the coordination problem [Haa95, Str95, NKCM94]. It refers to the coordination of authoring activities such that the synergy effect of cooperation can take place. The coordination of group activities strengthens the groups' efficiency by reducing communication and helps to avoid the occurrence of conflicting and repetitive actions.

On the other hand, a lot of authors like to work independently for periods of time [EMP⁺97, EM97, NKCM94]. They do not want to be disrupted by co-authors frequently. Authors want to keep their working documents *private* in a local workspace as long they are not in a “publishable state”. This implies that immediate propagation of document changes to the shared repository or directly to co-workers is often not desirable. This asynchronous working style is characterized by *alternating periods of individual and joint work*. This means that authors work largely independently, “syncing up” only when necessary to integrate results, or to reaffirm goals or plans [NKCM90, EMP⁺97, EM97]. Since it is neither possible nor desirable to fully preplan the authoring process, we have to accept that the appearance of some conflicts cannot be foreseen and, thus, the need to communicate and resolve conflicts can arise spontaneously.

To sum up, a cooperative transaction model that is well-suited for cooperative hyperdocument authoring has to support long-running, interactively controlled activities, in which resources can be used cooperatively. The emphasis is on the semantically correct sharing and exchange of documents. We will see that this is of equal importance for Design for Manufacturing and multi-user software development, but may be less important for workflow applications, where usually documents are passed along a predefined path among co-workers.

2.1.2 Design for Manufacturing

Design for Manufacturing (DfM) [VFSE95, TVF⁺95, TVL⁺98] can be seen as a variant of Concurrent Engineering. The scope of DfM is the engineering process of discrete complex industrial artefacts, usually separated into upstream processes (product design) and downstream processes (production realization, including engineering, planning, and manufacturing) [SRN93]. The essential part of DfM is the early involvement of specialists from downstream processes in the upstream design process. By involving production engineers, production plan-

ners, and manufacturers in the design process, this stage may take more time—but because problems in the design are detected earlier, fewer design cycles are necessary and the total development time is decreased.

As a result of the early involvement, some of the activities within DfM can, and usually will, be executed concurrently. Letting people work concurrently on shared and persistent artefacts implies that coordination and control are required. This means that frequent exchanges of (mostly small batches of) information are necessary, but it also means that group members need to work in a more systematic way. Thus, the strengthening of the design process by *overlapping* design phases requires extensive cooperation and coordination facilities. In comparison to cooperative authoring, more detailed knowledge about the engineering activities in different phases and about the sequence of processing is available. Therefore, DfM can be classified as a *structured design process*.

The information systems needed within the field of DfM are still in the developing phase, due to the complexity of the problems found in DfM [VFSE95, TVL⁺98]. Not only does DfM require the manipulation of complex data structures, it also requires sophisticated mechanisms of cooperative interactions to support the product development process. Within DfM, cooperation is found at different levels. It is found at the level of project management where people from different departments within a company have to work together, but it is also found at the level of designers that closely work together in the detail design of a certain part of the product.

In the detailed engineering stage of the design process, there is a need for versioning of the artefacts produced. If more than one version of an artefact is kept in the information system, versions can be divided into current, historical, and being-updated versions (*working copies*). In situations where cooperation support is needed, it is desirable that more than one working copy exists at the same time. Especially during concept design, different *alternative solutions* are studied in parallel. These working documents have to be *integrated* later on.

As each design process is of long-duration (or even *open-ended*) and involves a large number of deliberate and unconscious decisions, it is important to record the *history* of all decisions and actions undertaken by the users in the DfM process. This makes it possible to keep track of the *dependencies* among artefacts and eases the *revision of design decisions*.

2.1.3 Multi-User Software Development

Software development processes can be characterized as follows [WAN98, Kai95, CMG96, SCG96, WK96, BLW96, Lip92, BK91, Kai90]. First, they are interactive. A large part of their control is under the responsibility of human actors. Their execution is not predefined, but evolves incrementally as long as developers take decisions. This makes software development processes unpredictable, in the sense that their duration and effects are in many cases difficult to anticipate. Their duration can vary from a few days to months or even years. Also the individual actions executed by users, like editing sources, compilation, debugging, execution of test suites, etc., are usually of long duration.

Second, software development processes are usually iterative. They consist of repeated

activities like editing, compilation, testing, and debugging. This incremental approach to the production of complex systems allows to progressively converge to a product that holds the desired qualities. Naturally, decisions taken in this iterative processing need to be reversible to allow for the exploration of the design space of the software product.

Third, software development processes are jointly carried out by several users, since large systems typically require the concerted effort of many people to reach a semantically consistent state defined by, e.g., a successful build or regression test. For this, the software engineers need to exchange information during their work in a controlled way: they take profit of and influence the activities of other users working in parallel on the same (re)sources.

Usually, in multi-user software development environments (SDE) the exchange and sharing of information is realized using a persistent repository [Kai95, Kai93, KPS89, Hon88, KP87]. This repository stores all kinds of objects created during the software development project, such as source code, binaries, design documents, documentations, and test cases. Additionally, each user has his own *private workspace*, in which he can modify objects. For this, the objects first have to be read from the repository (check-out). Afterwards, the modified objects have to be written back to the shared repository (check-in) (cf. Section 2.3.4). Thus, cooperation is defined by the ability of users to access and produce intermediate “versions” of shared objects during their work. Of course, some form of synchronization is needed to coordinate access to the shared objects and to maintain consistency of the shared repository and the private workspaces. If co-workers are allowed to work simultaneously on the same objects in their private workspaces, additional mechanisms are needed for *merging* parallelly developed versions into a commonly accepted one [Hon88, Lip92, SK93, Kai95].

A subclass of SDE, called process-centered software development environments, [Kai93] provides additional coordination mechanisms through which a software development process may be partially specified—basically a partial order among software engineering tasks, constraints and obligations of those tasks, and the objects and tools used in these tasks. These systems also provide history management facilities to track the progress of the project. For an overview of software development environments, we refer to [Lip92, Kai95].

With respect to the spectrum of cooperative applications investigated, multi-user software development can be classified as a *semi-structured design process*, with a more or less predefined structure at the top-level and some known consistency constraints and termination criteria (like Design for Manufacturing). Possible activities are specification, implementation, and debugging which may be *overlapping and iterative*. At the lower abstraction levels the software development process is rather unstructured and cannot be specified in advance (like cooperative hypermedia document authoring).

2.1.4 Workflow Applications

Workflow applications [JLP⁺95, VPL95, TVF⁺95, TVL⁺98] are used to automate complicated business processes, e.g., to accomplish the production of goods or services. A *business process* is a set of one or more interconnected activities which collectively realize a business

objective or policy goal, normally within an organizational structure defining functional roles and relationships [WfM96a, WfM98]. A workflow application (for short workflow) is the automation of a business process, in whole or part, during which documents, information, or activities are *passed from one participant to another*, according to a set of *predefined rules* [WfM96a, WfM98]. Usually, workflow management systems (WfMS) [WfM96b, GHS95, AAAM97, CHRW98, Moh98] are used to define, create, and manage the execution of workflows.

Generally, a workflow consists of a collection of activities (also called tasks or steps) that are partially ordered by *control flow dependencies*. Additionally, there exist *data flow dependencies* that describe the flow of information among the different activities. Activities are the basic units of work and are usually processed by exactly one responsible actor, usually a human being. A workflow may also involve automated activities where an application program is used, but in most cases human beings are required to intervene and influence the flow of control and, hence, the flow of data or documents between the activities.

Compared to CDA, SDE, and DfM, workflow applications are characterized by well-defined business rules (inferred from organizational constraints), well-defined termination criteria, and less interactive processing. Workflow application focus mainly on the *coordination* of activities and less on the cooperative sharing of data or documents. Like CDA, SDE, and DfM, workflow applications are long-running processes where, dependent on the type of the workflow application, a workflow instance might last from several minutes up to months or even years. Therefore, it is essential to keep track of the progress of the workflow. This is especially important for auditing purposes, exception handling, and compensation of (parts of) a workflow.

2.1.5 Common Characteristics and Differences

In this section, we summarize the common characteristics of the application scenarios we described. In general, we can identify a spectrum, ranging from cooperative hyperdocument authoring to workflow applications, where the process becomes more fixed and prescribed, allowing a more deterministic description of the problem-solving process and termination conditions. Figure 2.1 illustrates this spectrum and characterizes the application scenarios we investigated.

All of the investigated scenarios can be classified as *asynchronous* cooperative applications, emphasizing different aspects of communication, cooperation on shared data, and coordination [TVL⁺98, EGR91]. Asynchronous cooperative work does not require that all co-workers are present at the same time. Co-workers work independently for periods of time, and there is little need for immediate propagation of results and fine-grained coordination. In all cases, some kind of a persistent repository is involved to enable the sharing and exchange of data among the users. In all of the scenarios, we can identify a strong need for *consistency*—both for shared data and for private data.

In particular, one can identify the following common characteristics of the investigated

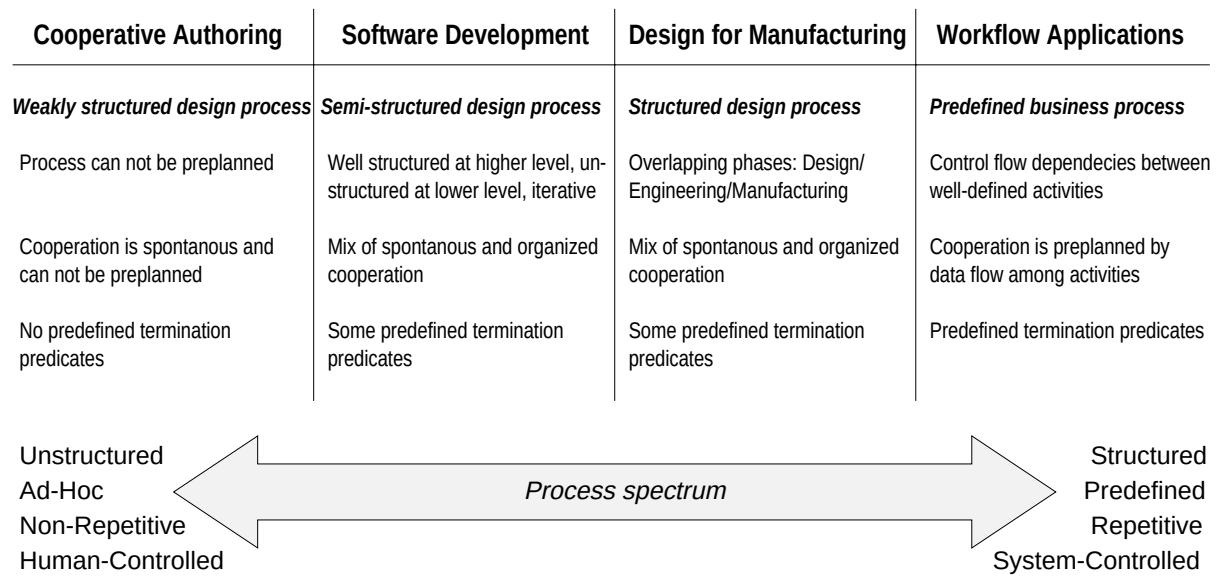


Figure 2.1: Spectrum of Investigated Classes of Cooperative Applications.

classes of cooperative applications (for more details, we refer to [TVL⁺98, WAN98, TVF⁺95]):

1. Activities are processed interactively by human beings.
2. Activities are usually of long and uncertain duration.
3. Not all co-workers may be simultaneously present in the cooperative process.
4. Cooperation is characterized by alternating periods of individual and joint work.
5. Cooperation is based on the exchange and sharing of information among co-workers.
6. Data and documents processed in the cooperative process are persistent.
7. Shared and private workspaces co-exist.
8. Decisions and actions taken by the co-workers are recorded.
9. It is likely that decisions and actions made during the cooperative effort are revised.
10. There may exist application-specific correctness criteria, both for the work of a single user and for the overall cooperative effort.
11. Between activities there may exist control flow dependencies that are derived from the application domain.
12. It is likely that co-workers are geographically distributed and only partially connected, due to the increasing mobility of co-workers.

Of course, the relevance of these characteristics varies for the different application domains. For example, in workflow scenarios, control flow dependencies are of higher importance than in the cooperative authoring domain (cf. Figure 2.1).

2.2 Requirements for a Cooperative Transaction Model

In the following, we present the requirements for a cooperative transaction model suitable for a broad range of cooperative applications. Since the primary goal of a cooperative transaction model is to synchronize access to persistent data shared among multiple users, we concentrate in the following on data management aspects, rather than on the organizational aspects found in cooperative work.

2.2.1 Illustrating Example

We introduce a simple example from the software development domain to illustrate our requirements for a cooperative transaction model. We have chosen this example because most of the readers are familiar with software development and because similar examples are often used to illustrate advanced transaction models, e.g., in [BK91, Kai95, RC97].

Suppose, a group of programmers is cooperatively working on the development of a Java application program, using a DBMS-based software development environment (SDE). The SDE encapsulates Java application programs, Java packages (which are similar to class libraries), and Java classes as database objects. By providing a well-defined interface to these objects, the SDE abstracts from the internal details of the structure and storage of objects. Within the SDE, a Java program consists of several class objects and packages. A package object consists of several class objects, sub-packages, and programmer's annotations to the package. Finally, class objects consist of the class' source code, the class' bytecode, and again textual annotations. Examples of SDE operations are browsing/modifying the source code of a class, reading/adding/removing annotations of a class, and compiling the class. To give programmers convenient access to the objects managed by the SDE, the objects can be stored internally as inter-related objects and visualized by a hypermedia desktop, like in [BLW96].

Example 1 (Software development example) During the development of an application program some bugs occur. Alice and Bob are assigned the task to fix the bugs. Alice should fix a bug suspected in class X while Bob's task is to fix the bugs in class Y. Both classes use methods of package Z.

Alice starts a new debug session and begins to edit class X while Bob starts a “transaction” to fix the bugs in Y. During debugging Bob detects that some bugs in class Y are possibly caused by bugs in package Z. Therefore, he starts to work on Z. Although Bob's modifications to Y do not affect class X it may be the case that the modification and re-compilation of Z may introduce inconsistencies with class X. □

As discussed in [BK91, Kai95] the usage of concurrency control policies that apply serializability as their correctness criteria prohibits Alice and Bob to cooperatively work on package Z. Even concurrently adding and reading annotations to class objects is not possible. The reason is that serializability does not allow users' transactions to influence each other in a controlled way.

2.2.2 Consolidated Requirements

Based on our running example, we now present the consolidated requirements for a cooperative transaction model [TVL⁺98, WAN98, AKT⁺96]. The full, detailed discussion of the requirements for the cooperative transaction model and an overall coordination model (including its specification language) can be found in [TVF⁺95, VTF⁺95].

2.2.2.1 Relaxed Atomicity [Requirement (A)]

Since cooperative processes are long-lived and interactive, the rollback of the whole cooperative work process in the case of a failure is generally not acceptable. It is required that a cooperative process should be able to proceed and eventually succeed even if parts of it fail. A failure within one user's activity should not imply the rollback of another user's work in their joint effort. Moreover, the loss of work should be minimized if a failure occurs.

The specific atomicity constraints appear to be application-dependent, i.e., they have to take into account the specific semantics and the structure of the cooperative work process. System-initiated transaction aborts should be dealt with at a lower level. Traditional short atomic transactions can form the basis for this and provide the basis for persistency and durability of data processed.

Example 2 (Relaxed Atomicity) If a failure occurs within Bob's debug transaction, *not* the whole cooperative debugging session, i.e., Alice's *and* Bob's transaction, should be rolled back—even if Alice has been able to observe Bob's modifications to package Z.

Furthermore, if the failure occurs while Bob re-compiles package Z, only the bytecode of Z should be restored to the state before the compilation. Bob's modifications to the source code of Z need not to be undone since they are not affected by the failure. □

2.2.2.2 Support for Retraction of Decisions [Requirement (B)]

To support interactive control of activities, a cooperative transaction model has to provide services that allow to retract decisions taken by the cooperating actors. Hence, the model has to capture the relevant dependencies between users' actions by exploiting their semantics. Additionally, the transaction model has to provide means to specify how certain activities and decisions can be compensated for.

This allows a user to explore several alternatives to a problem and to revise erroneous actions. It is important to keep track of the progress of a cooperative effort, i.e., all relevant

actions and decisions taken by the co-workers should be logged in a history together with their dependency structure. This history can also be used to coordinate the work of multiple users and to enforce the correct control flow in the cooperative effort.

Example 3 (Retraction of Decisions) For instance, if Bob modifies some classes in package Z and then recompiles Z without success, he might detect that his modifications introduced new errors. In this case, he might want to revert some of his actions that changed the source code of Z.

Or, suppose that after Bob commits his debug transaction, Alice browses the history of Bob's changes and detects that Bob has changed a constructor method in package Z that is often used in class X. If she wants to undo Bob's changes, the transaction model should be able to rollback to a consistent system state, i.e., if Bob's modifications to the source of Z are undone the bytecode of Z also has to be automatically restored to the old state. □

2.2.2.3 Support for Consistent Exchange of Work [Requirement (C)]

The sharing and exchange of final and of intermediate results among co-workers is a prerequisite for cooperation. A cooperative transaction model should provide mechanisms to facilitate the exchange of tentative or partial results among co-workers, while at the same time guaranteeing that no anomalies are introduced by the exchanges. Hence, it must be possible to detect when two actions of different users are in conflict and to semi-automatically resolve such conflicts.

The transaction model should offer well-defined primitives for the semantically correct exchange of information between co-workers. The exchange facilities should be adjustable in order to support information exchange on behalf of human beings and also on behalf of the cooperative application. This implies that the exchange primitives can be directly used by co-workers within the cooperative process, but there should also be means to couple them with additional coordination facilities realized on top of the transaction model.

Example 4 (Consistent Exchange of Work) Suppose, Alice wants to browse the source code of package Z while Bob's debug transaction is still ongoing (to check if Bob's modifications affect her work). For this, she needs a mechanism for controlled access to package Z that is currently under modification by Bob. Similarly, Alice might wish to annotate class X and make her annotation *visible* to Bob while her transaction is still running.

Or, assume Alice is allowed to test and debug class X using the new bytecode of package Z generated by Bob. To guarantee that no anomalies are introduced, the transaction model should give Alice access either to *both* the source and the bytecode or *only* to the new source code of Z—but not to only the new bytecode, because the new bytecode is dependent on the modified source code and no longer compatible with the old source of Z that Alice has seen. □

2.2.2.4 Support for Private and Shared Data [Requirement (D)]

To explore different solutions of the same problem, different co-workers should be able to work at the *same* time on the *same* data. To avoid interference from co-workers and to allow for asynchronous work, the cooperative transaction model should be able to manage private copies of data objects in separate workspaces. Upon a user's request, it should be possible to exchange "working versions" and to combine them into a commonly accepted one, shared among some or all users.

To reduce the number of conflicts during re-integration, the transaction model should be able to exploit the semantics of the operations executed on different private "versions" of the objects. Additionally, appropriate conflict resolution mechanisms have to be provided by the transaction model.

Another reason for the separation of shared and private data is the possible geographic distribution of co-workers and the needed mobility support. The transaction model should be able to deal with multiple copies of data objects to provide better response time and to enable work of users, while they are disconnected from the network. Of course, conflicts have to be detected automatically during reconciliation of work at re-connect. The necessary conflict resolutions should be guided by the cooperative transaction model.

Example 5 (Private and Shared Data) For instance, Alice should be able to use the old version of package Z for her tests while Bob is modifying and recompiling Z, if they decide to *temporarily* allow inconsistencies which will be resolved at a later point in time. Even if Alice wants to modify package Z, this should be allowed at the price of *some* modifications to be undone later to produce a consistent version set of X, Y, and Z.

In case Alice wants to debug class X while she is on a business trip, she has to reserve and copy the whole package Z onto her laptop in order to be able to test and debug X. This should not preclude Bob from working on his stationary copy of Z. □

2.2.2.5 Coordination of Individual and Joint Work [Requirement (E)]

In cooperative applications different users work together to achieve a common goal. Coordination facilities are useful to guide each user's work towards this goal. To describe the pre-planned parts of cooperative processes, the decomposition of the overall work, and the control flow dependencies, a cooperative transaction model should allow the definition of *execution constraints* by additional coordination facilities.

Execution constraints should be able to describe the possible structure of both a single user's activity and the overall cooperative work process. This includes constraints on the occurrence of particular subactivities or user actions and on their execution order within the overall cooperative process.

Example 6 (Coordination of Individual and Joint Work) An example for an execution constraint for a cooperative software development process is to prohibit the modification of the

design document while the development process is solely in the testing phase. This does not preclude that the design phase is activated again (if allowed by the execution constraints). An example for an execution constraint at the level of a single user's activity is that Bob can only “commit” his debug transaction if he has successfully recompiled all modified classes. □

It appears that the kind of execution constraints needed is heavily dependent on the particular cooperative application. Moreover, execution constraints are usually imposed by organizational aspects of the work process rather than by correctness concerns for data sharing and exchange. For these reasons, we decided that the coordination facilities need not be part of the transaction model itself. Instead, the transaction model should offer appropriate interfaces that can be used by an extra coordination model built upon the transaction model.

2.2.2.6 Requirements Outside the Scope of Cooperative Transaction Models

Further requirements are imposed on computer system support for cooperative applications. For example, to support mutual awareness of co-workers, some kind of session control and notification mechanisms are needed. Additional communication facilities, like e-mail or audio/video conferencing channels, may be required for direct negotiation among co-workers.

Such services are not covered by a cooperative transaction model, since it aims at synchronizing the cooperative usage of persistent, shared data. Nevertheless, a cooperative transaction model satisfying the requirements discussed above can provide an application-independent nucleus to build a wide spectrum of cooperative applications on top of it.

2.3 Transaction Support for Cooperative Applications: An Overview

The issue of synchronizing work of multiple cooperating actors on shared data is treated in several fields of computer science, including advanced transaction models, groupware, and workflow management systems.

To support specific aspects of cooperative applications, several advanced transaction models have been proposed in recent years. In the following, we give a detailed overview of those that were particularly relevant for developing the COACT cooperative transaction model described in this thesis. We review how these models fulfill the requirements presented in the previous section. More related work will be discussed in following chapters of this dissertation where appropriate. For additional information and for a discussion of other advanced transaction models we refer the reader to [VWP⁺98, ELMB92, Elm92, Hsu93, BK91, Kai95, RC97]. We conclude this chapter by an overview of groupware systems and workflow management systems and discuss their main ideas and limitations with respect to the applications described in Section 2.1.

2.3.1 Semantics-Based Concurrency Control for Abstract Data Types

Usually, the semantics of shared data and operations in cooperative applications is much more complex than the read/write semantics used by the traditional transaction model. As a result traditional concurrency control here is more restrictive than necessary, as it does not take into account the semantics of the data objects accessed [RC97, VWP⁺98]. *Semantics-based concurrency control* was first proposed in the context of abstract data types [Kor83, SS84, Wei88a, GM83] and later adopted for object-oriented DBMSs [RGN90, AAS93, MRW⁺93]. Its main goal is to achieve a higher degree of concurrency than in the traditional read/write model. Additionally, the semantics of data objects can be used to increase the availability of data and provide more flexible recovery. For an extensive discussion of semantics-based concurrency control see [RC97].

The idea behind semantics-based concurrency control is as follows. By abstracting from the low-level details, i.e., the concrete implementation in the database, and by exploiting the high-level semantics of data objects and operations, we can ignore certain conflicts that might occur between decompositions of the operations into elementary read/write sequences. The most commonly used approach to capture the semantics of data objects is to specify *commutativity relations* among operations defined on single data objects. Informally, two operations commute, i.e., do not conflict, if their effects on the state of the object and their return values are the same regardless of their execution order. This property ensures that it is not possible to observe a difference between both execution orders in the system. When a transaction requests the execution of an operation, this request can be granted if the operation commutes with all other operations of uncommitted transactions. This policy ensures that concurrent transactions are serializable at a semantic level and that (generalized) ACID properties of transactions are preserved. The prerequisite is that the single operations are atomic.

Example 7 (Semantics-based concurrency control) Consider the following operations to add and remove annotations to a class object in the example software development scenario (cf. Example 1 on page 21). The operation `annC(C:Class):AnnID` is used to annotate a class `C`; it returns the identifier of the newly created annotation (`AnnID`). The operation `remA(C:Class,A:AnnID)` is used to remove the annotation with the specified identifier from class `C`.

Assume, two `annC(X)` operations add different annotations to class `X`. Both operations modify the class object `X` in the SDE. Hence, they conflict on the level of read/write operations. However, by considering the high-level operations semantics, both operations commute, since the resulting set of annotations for class `X` is the same in both execution orders. $\square$

Weihl [Wei88a, Wei89a, LMWF94] distinguishes between *backward commutativity* and *forward commutativity*. The difference is the underlying execution model. The former is applicable to an update-in-place execution model whereas the latter is applicable to deferred-update models. Other (non-symmetric) semantic relations have been proposed in the literature that relax the requirements on the state equivalence of the data objects and enable an even higher

degree of concurrency control. In [Her86, Her90], the *invalidated-by* relationship has been introduced for optimistic concurrency. *Recoverability* [BR92] allows non-commuting but recoverable operations to be executed concurrently.

Semantics-based concurrency control fulfills none of the requirements stated in Section 2.2 directly, but achieves a higher degree of concurrency. Thereby, it reduces the probability of long waits or aborts which is important when transactions are of long duration. Commutativity relations are also a major building block of the COACT model described in Section 3.

2.3.2 Multi-Level and Open-Nested Transactions

The basic goal of multi-level transactions [Wei86, Wei88b, BSW88, Wei91, WH93, WS92, DSW94] and its generalization to open-nested transactions [WS92, MRKN92, MRW⁺93, WDSS93, MR93] is to enhance inter-transaction parallelism and to allow for a finer recovery granularity. This is achieved by modeling a transaction as a tree of subtransactions, by exploiting the semantics of database operations, and through recovery by *compensation* rather than by state-based undo.

A *multi-level transaction* has an implicit internal structure where the nodes in the transaction tree correspond to executions of database operations at particular levels of abstractions in a layered DBMS. Both subtransactions and actions (at the leaves of the tree) are considered to be atomic. All multi-level transaction trees have to have the same depth. *Multi-level serializability* [Wei86, Wei91] replaces traditional read/write serializability as correctness criterion while preserving generalized ACID properties of transactions [VWP⁺98]. The idea is to exploit the semantics of database operations in level-specific conflict relations based on commutativity. Commutativity at a specific layer refers to abstract database states and not to concrete states, i.e., operations at a lower layer are allowed to see differences. Conflicts at a lower layer can be ignored if two operations at a higher layer commute with respect to the abstract state. This policy allows more concurrency than “flat” semantics-based serializability (cf. Section 2.3.1) [VWP⁺98].

For *open-nested transactions*, transaction siblings in the transaction tree are allowed to have different nesting depth, like in closed-nested transactions [Mos82, HR87b, HR87a, TW97]. As opposed to closed-nested transactions, open-nested transactions are based on the idea of early visibility of subtransactions in multi-level transactions. The concept of *semantic serializability* [MRW⁺93, Rak97] generalizes multi-level serializability to arbitrary nesting levels of transactions. Open nested transaction models are well-suited for object-oriented DBMSs [WS92, MRW⁺93, MR93] and federated DBMSs [WS92, Mut94, DSW94]. The dynamic method invocation hierarchy can be transformed to an open-nested transaction.

As a consequence of the early visibility of modifications to commuting subtransactions, the abort of a subtransaction can no longer be implemented by restoring the database state directly to the state before the subtransaction. An already committed subtransaction S has to be undone by its *compensating subtransaction* S^{comp} that semantically undoes the effects of S . Before a subtransaction S can be compensated, all other subtransactions of the same transac-

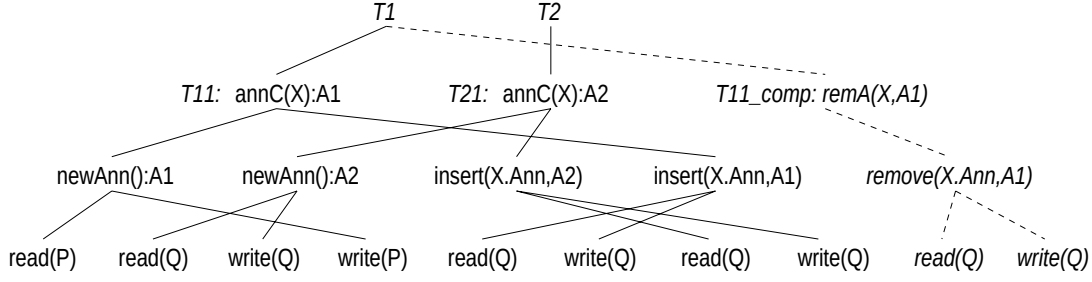


Figure 2.2: Example of Open-Nested Transactions.

tion executed after S have to be compensated in inverse order. This is similar to the approach used in SAGAs [GMS87, GMGK⁺91, CR92]. The compensating subtransactions are fully embedded in the concurrency control mechanism, i.e., they are executed as regular transactions in this respect [Wei91, AVA⁺94]. To enable compensation, the semantic lock for the compensating subtransactions has to be granted before the execution of a specific subtransaction [WHBM90]. This ensures that the compensating subtransaction will be executable if needed, and the atomicity property of transactions is thus preserved. The need to prepare for possible compensation when a subtransaction is executed limits the potential degree of concurrency achievable [VWP⁺98].

Example 8 (Open-nested transactions) Figure 2.2 shows an example of open-nested transactions. Alice starts transaction T_1 and annotates class X while Bob starts transaction T_2 and annotates the same class simultaneously. Each $\text{annC}(X)$ subtransaction (cf. Example 7) creates a new annotation and inserts it into the set of annotations associated with class X , i.e., it executes newAnn and insert subtransactions. The leaves of the tree consist of atomic read/write operations. In contrast to “flat” semantic concurrency control (cf. Example 7), it is now possible to execute the $\text{annC}(X)$ operations concurrently. Clearly, the schedule at the leaf level is not read/write serializable but T_1 and T_2 are semantically serializable (and also multi-level serializable since each branch in the transaction tree has the same depth).

If Alice aborts her transaction later on, the system has to execute the compensating operation for T_{11} to remove her annotation from class X , i.e., $T_{11}^{\text{comp}} : \text{remA}(X, A1)$. This is possible without affecting Bob's transaction since T_{21} and T_{11}^{comp} commute (or T_{11} and T_{21} commute). State-based undo is not possible because Bob's annotation would have been removed from class X , too, in this case. $\square$

Long running transactions are supported by multi-level and open-nested transactions because of the high degree of parallelism. In addition, these models provide a finer granularity of recovery units (partial rollback). Nevertheless, sharing of tentative data among co-workers executing concurrent transactions is not possible because both multi-level and open-nested transactions stick to a variant of serializability as their correctness criteria. User-initiated undo can be implemented by exploiting the commutativity relations and the compensation mechanism

(cf. Section 3.5.3). In [WS92] some further extensions of open-nested transactions are outlined to adapt them to specific application needs. In general, we can say that requirements (A) and (B) (cf. Section 2.2) are partially supported by multi-level transactions and open-nested transactions.

In the following, we discuss advanced transaction models especially developed for cooperative environments.

2.3.3 Dynamic Restructuring of Transactions

The *split/join transaction model* [PKH88, KP92] was developed for open-ended activities characterized by uncertain duration, unpredictable developments, and interactions with other concurrent activities. The model addresses cooperation among users by supporting the dynamic restructuring of ongoing ACID transactions. A *split* command allows to split a running transaction into two transactions while a *join* command allows to incorporate two transactions into one transaction. The correctness criterion used in this model is read/write (conflict) serializability of transactions to be committed. In contrast to other models, there exists no simple relationship between the set of transactions initiated and those that are committed. Serializability is guaranteed by permitting only those restructuring commands that preserve a serializable history.

A *split* divides a running transaction T_A into two transactions, T'_A and T_B . Read and write operations that have been executed by T_A are assigned to T'_A or T_B . T'_A and T_B are then responsible for committing or aborting the assigned operations. A split is only allowed to be executed if T'_A and T_B are serializable. Depending on the assignment of conflicting operations, we can distinguish between a *serial split*, where the commit of T_B is commit-dependent [BOH⁺92, CR92] on T'_A , and an *independent split*, where T_B and T'_A can commit and abort independently. For interactively executed transactions, the serial case is only allowed if T_A is committed immediately after the split operation.

The *join* command joins a transaction T_A with the target transaction T_B by adding the read and write sets of T_A to T_B . Before the join becomes valid and T_A disappears, T_B has to accept it. Afterwards, T_B can use any resources from T_A , and all changes of T_A to the database will become valid at commit of T_B . Uncontrolled joining of two transactions can lead to non-serializable histories. For example, if T_A is serialized before T_C , T_C is serialized before T_B , and T_A and T_B are joined, then the resulting “merged” transaction $T_{A \cup B}$ is not serializable with respect to T_C .

Example 9 (Split/join transactions) An example of split/join transactions is shown in Figure 2.3. Alice and Bob start their own debug transactions. Bob modifies the source of class Y (denoted by X.S in Figure 2.3) and recompiles it, resulting in a change of the bytecode X.B. Alice does the same for class X. Assume, Bob wants to access the source of class X. This is not possible because the source and bytecode of X are locked by Alice (if locking is applied). Therefore, he asks Alice to release the source of X. Alice fulfills his request by splitting her

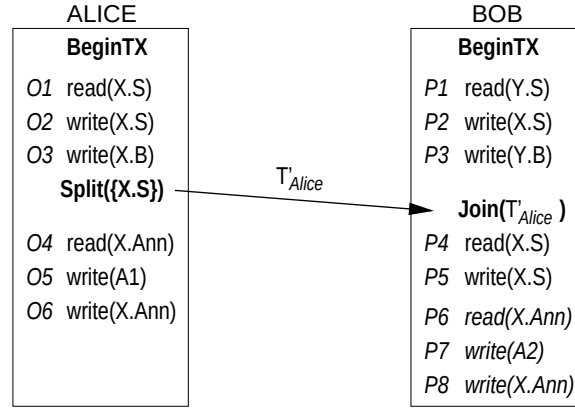


Figure 2.3: Example of Splitting and Joining Transactions.

ongoing transaction ($\text{Split}(\{X.S\})$). This results in the creation of a new transaction T'_{Alice} (independent case). The read and write sets of T'_{Alice} consist of the source of X while the read set of the original transaction becomes empty and the write set becomes $\{X.B\}$ (the bytecode of X). Bob joins T'_{Alice} to his running transaction. Now, Bob is able to modify the source of X, but Alice is no longer able to browse or modify the source of X because T'_{Alice} is now under Bob's control.

Assume, Alice annotates class X (operations o_4, o_5, o_6 in Figure 2.3) and Bob tries to annotate X, too. This results in blocking Bob's transactions because operation p_6 conflicts with o_6 (note that this conflict does not occur if the semantics of operations had been exploited). If Alice splits her transaction again ($\text{Split}(\{X.Ann\})$) and commits the new transaction immediately, Bob can continue because the lock on X.Ann is released now. Note, if Bob now reads the set of annotations, he will discover a new annotation but he is not able to read it, since the annotation A1 is still locked by Alice (the remaining part of T_{Alice} , i.e., $\text{write}(A1)$, is not yet committed). $\square$

A concept similar to split/join, i.e., *delegation*, can be found in ACTA [CR90, CR91, CR92, CR94]. ACTA is a transaction meta-model that can be used to specify the types of dependencies between transactions and within transactions (cf. Section 2.3.6). One of ACTA's building blocks is the delegation primitive [CR93]. Delegation in ACTA means that the responsibility for committing or aborting actions can be delegated from one transaction to another transaction to broaden the visibility of the delegatee and to tailor the recovery properties of a transaction model. In this way, tentative results can be made visible to co-workers.

Splitting and joining ongoing transactions enable a cooperative behavior by delegating parts of transactions to a co-worker. Thus, the split/join transaction model fulfills requirement (C) quite well. Other advantages of the split/join model are adaptive recovery (requirement (A)) and increased concurrency if the splitted transaction is committed immediately, i.e., the resources are released and made persistent prior to the commit of the original transaction. An

open problem is how to ensure that two transactions resulting from a split form again a consistent unit of work. The requirements (B), (D), and (E) (retraction of decisions, private work areas, and coordination of joint work) are not addressed by the split/join transaction model.

2.3.4 Check-Out/Check-In, Versioning, and Workspaces

One of the simplest forms of synchronizing the access of a team of designers to a shared repository is the *check-out/check-in model* [Roc75, KW84, Tic85, Kat90, KBGW91, Kai95]. This model has been implemented, for example, by widely used version control tools like SCCS [Roc75] and RCS [Tic85]. Data objects have to be explicitly checked-out by a designer, i.e., objects are copied from the shared database to the *private workspace*, where the data objects can be manipulated. In the simplest case, checked-out data objects are reserved for exclusive access, until they are later checked-in into the shared database. Check-out and check-in operations have to be atomic. Obviously, this policy limits the availability of data objects enormously. Thus, the basic check-out/check-in model can hinder cooperation, since each user works separately in isolation [BK91, Kai95]. Therefore, the check-out/check-in model appears in most cases in tandem with versions and configurations [Kat90] like in version control tools such as RCS [Tic85].

A *version* of an object is simply a “snapshot” of an object's state at a particular point in time [Kat90, Kai95]. Versions may be mutable or immutable. If an immutable version is checked-out, a new successor version is first derived from the original one and then copied to the workspace. The checked-out version is reserved exclusively, but another user may derive a new version that can be checked out. This results in branches in the version history of an object. Of course, the parallel versions have to be merged (manually) in order to come up with a single, commonly agreed result of the object. Only some systems (most notably, NSE [Hon88] and Biin [SCG⁺89]) provide tools that aid in merging multiple parallel versions to the same data objects [Kat90, BK91, Kai95].

Another problem is to keep track of which versions of different data objects form a consistent set of versions. For example, if there exist several versions of class objects X and Y, it is difficult to find out which versions of X and Y can be used to compile executable code. Therefore, versions are grouped into consistent versions sets, called *configurations* [LC87, WBHN87, WBMN88, Kat90]. Dealing with configurations complicates version management and check-out/check-in. Additionally, this can decrease concurrency because entire configurations have to be reserved always.

Example 10 (Check-out/check-in) An example is shown in Figure 2.4. Alice checks-out class X, i.e., the source and the bytecode of X and the annotations made to X. Bob checks out class Y. This leads to exclusive locks for X and Y in the public database. Alice edits the sourcecode of X (operations o_1 and o_2), compiles the bytecode (o_3), and annotates class X (o_4, o_5, o_6). Bob edits and compiles Y (operations p_1, p_2 , and p_3). He recognizes that he needs class X to check if his modifications affect this class. Check-out of X is not possible because it

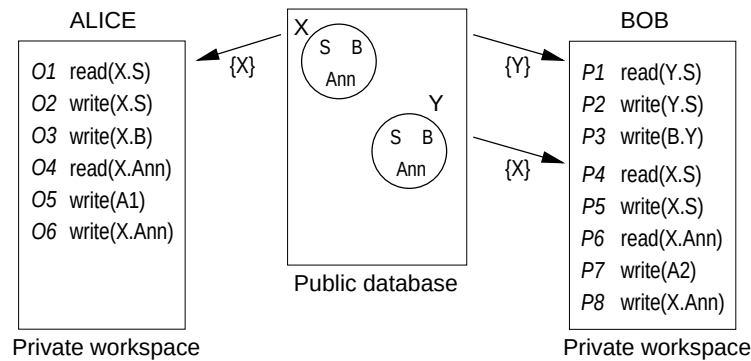


Figure 2.4: Example of Check-Out/Check-In.

is exclusively reserved by Alice. He can either create a new version of X that he can check-out, or ask Alice to check-in X. In the former case, Bob cannot see the Alice's modifications to the sourcecode, not even the annotations Alice made to X. The parallelly developed versions of X have to be merged manually later on. Even if Bob has only annotated class X an automatic merge would not be possible, because the check-out/check-in model does not take into account the (semantics of) operations executed in the private workspaces. In the latter case, Alice is no longer able to work on X, but Bob can observe her modifications.

Other problems will arise if Alice is allowed to check-out/check-in only the sourcecode or the bytecode of X. In this case, a class is no longer a (implicit) configuration. This would enable a higher degree of cooperation but might violate consistency. For example, if Alice checks-out the source and bytecode of X (now separately) and later checks-in the bytecode, Bob can observe inconsistencies (he might not be aware of) between his version of the source and the bytecode of X. $\square$

For the purpose of software development, several extensions of the check-out model have been developed, which take advantage of the possibility of software consistency checking in those environments [KF87, KP87, Hon88, KPS89, Kai95]. The Network Software Environment (NSE) [Hon88] supports an optimistic coordination scheme by forcing users to merge concurrently created versions of sourcecode, also known as *copy/modify/merge* cycle. Smile [KF87] provides one shared repository for the public configuration and a private workspace with a private configuration for each user. If a private configuration is checked-in, the system recompiles the modified source files and tries to link them to the unmodified modules in the shared repository. If this fails, the check-in operation is aborted. Infuse [KP87, KPS89] generalizes this approach by supporting additional semi-public workspaces and by using execution-time consistency checking. Due to the fact that such consistency checking mechanisms are not available in other application domains, these approaches do not provide a general solution.

Obviously, check-out/check-in models fulfill requirement (D) stated in Section 2.2, i.e., support for private and shared data. If check-out/check-in models are combined with versioning

mechanisms and configurations, they fulfill the requirements (A), (B), and (C) to some extent. Versioning supports the exchange of results among co-workers, if one accepts that versions have to be (manually) merged later and that version sets may become inconsistent. Retraction of decision and relaxed atomicity is partially supported by versioning, too, because one can “rollback” to immutable versions in the version history.

2.3.5 Group-Oriented Transaction Approaches

Design transactions [KLMP84] and *CAD transactions* [BKK85, KKB87] enhance the basic check-out/check-in model by introducing a hierarchy of a public database, several semi-public databases (one per design project transaction), and private databases for active designers' transactions. Kim et al. [KLMP84] apply a locking protocol with extended lock modes to enable the exchange of data objects among private databases via their associated semi-public database. CAD transactions enable cooperation among designer's transactions through a client/subcontractor model using predicate-wise serializability as correctness criteria [BKK85, KKB87].

Similarly, most other group-oriented transaction approaches describe the overall working process as a transaction hierarchy consisting of group transactions. Individual user transactions form the leaves of the transaction hierarchy. Visibility between transactions is supported by extended locking schemes [KSUW85, FZ89, VEH89, US92] or by following predefined access patterns that define the application-specific correctness criteria like in [NZ90, NRZ92]. The *CONCORD model* [RMH⁺94, MHR96, RMH96, Rit97] explicitly supports different kinds of cooperation relationships that allow to share data among (hierarchically structured) design activities.

In the *cooperative transaction hierarchy* approach [NZ90, NSZ91, NRZ92] each group transaction defines a set of rules that specify the way the child transactions can interact and how data objects can be shared among them. Correct histories are specified by *patterns* and *conflicts* [Ska89, Ska91] using finite-state automata. Conflicts are comparable to locks in the sense that they specify when certain operations cannot be executed. Patterns specify that specific operation sequences have to occur. Cooperation is enabled by allowing non-serializable executions of transactions that belong to the same group. The usability of this model for cooperative applications is restricted since significant parts of the cooperative application have to be pre-specified (by means of patterns and conflicts) in order to describe whether a particular non-serializable execution is correct or not. Thus, support for interactive user-control of transaction and revisions of decision is limited. Relaxed atomicity is supported by means of pattern dependencies [NRZ92].

The *participant transaction model* [Kai90, BK91, Kai95] defines each transaction as a participant in a specified *domain*. The domain represents the set of user transactions controlled by users collaborating on a common task. Participant transactions in the same domain need not be serializable—only transactions of different domains have to be serializable. As the participant transaction model is based on read/write actions and no concurrency control applies to a domain, this can lead to inconsistencies of data accessed in a domain (e.g., updates based on data

no longer valid) and counter-intuitive behavior of the system from the user's viewpoint. Moreover, support for relaxed atomicity, user-initiated revision of work, and for private and shared data is limited. Coordination of joint and individual work (requirement (E)) is not addressed in the participant transaction model.

2.3.6 Transactional Workflow Models

The area of *transactional workflow* approaches¹ [SR93, RB94, RS95b, WS97, ANRS92, SANR92, ARSS93, JNRS93] usually comprises specification languages to express various kinds of *execution constraints* for a set of (transactional) tasks. This can be done either by supporting a script language like in the ConTract model [WR92], by a declarative specification of the execution structure in terms of externally visible execution states [ARSS93], or by ECA rules like in ATM [DHL90]. In the following, we briefly describe some of the approaches in the area of transactional workflows.

SAGAs [GMS87, GMGK⁺91, CR92] have been proposed to deal with long-lived activities. Basically, a SAGA consists of a sequence of ACID transactions that are allowed to interleave with component transactions of other SAGAs. However, the entire SAGA can be made atomic by compensating all committed component transactions in *inverse order* (cf. Sections 2.3.2 and 3.5.3). SAGAs relax the isolation property of transactions and increase inter-transaction concurrency. An extension allowing the nesting of SAGAs has been proposed in [GMGK⁺91, CR92].

ConTracts [Reu89, WR90, WR92, Sch93, RS95a, SR96] share the basic idea with SAGAs, but allow to explicitly specify control and information flow between their predefined component transactions (*steps*) by means of a *script*. The scripting language provides language constructs for sequential and parallel execution, as well as for branching and looping. Similar to the NT/PV model [KS88, KS90, KS94a], the ConTract model uses *invariants*, i.e., consistency constraints, defined on the database for concurrency control purposes to relax isolation. One emphasis of the ConTract model is *forward recoverability*, i.e., in case of a failure the state of the ConTract is restored and its execution may continue. This is achieved by using private context databases where the state of an entire ConTract is checkpointed. Moreover, ConTracts provide relaxed atomicity so that a ConTract may be interrupted and re-instantiated.

Also, *Flex transactions* [ELLR90, RELL90, Leu91] allow the early release of resources. A Flex transaction consist of a set of tasks with a set of functional equivalent subtransactions for each. A set of execution dependencies on the subtransactions, including failure dependencies, success dependencies, or dependencies on external events can be specified. In [KPE92], parallel logic programming was used for specification. To relax isolation, Flex transactions use compensation. Global atomicity is relaxed by allowing to specify acceptable *termination states* in which some subtransactions may be aborted.

¹The term *transactional workflow* was introduced by Sheth and Rusinkiewicz [SR93] to clearly recognize the relevance of transactions to workflows. It has been subsequently used by a number of researchers, e.g., [BDS⁺93, KS95b, GHS95, TV95].

The *Activities/Transaction Model* (ATM) [DHL90, DHL91, DS93] allows to integrate both ACID transactions and so-called activities which do not need to be ACID. A long running activity is modeled as a set of execution units that may consist of other activities or transactions recursively. Control flow and data flow may be specified statically in a *script* or dynamically by event-condition-action rules. Recovery is supported by rule-based specification of compensation transactions and *contingency transactions* which represent alternatives.

Enforcement of *intertask dependencies* in transactional workflows is discussed in [ARSS93, RS95b, RB94, GRL94]. Tasks are modeled by providing their (visible) states together with significant events corresponding to the state transitions (start, commit, abort, etc.). Intertask dependencies can be classified as enforceable, rejectable, or delayable. In [ARSS93], intertask dependencies such as the order dependencies $e_1 < e_2$ or existence dependencies $e_1 \rightarrow e_2$ [Kle91] between significant events are formally specified using computational tree logic. The *METEOR* system [KS94b, KS95b, SKM⁺96b, SK98] is also based on the concept of intertask dependencies.

The *Transaction Specification and Execution Environment* (TSME) [GHKM94, GHM⁺93, GH94, GHM96] is a programmable system that supports the specification of transactional workflows and the configuration of transaction managers to enforce the desired models. It provides a specification language to express various kinds of intra-transaction and inter-transaction dependencies and the correctness criteria that the transaction must satisfy.

ACTA [CR90, CR91, CR92, CR94] is a *transaction meta-model* to specify advanced transaction models and to reason about (1) the effects of transactions on data objects and (2) the interactions among transactions. ACTA has five building blocks: histories, transaction dependencies, the view of a transaction, the conflict set of a transaction, and delegation (cf. Section 2.3.3). The transaction models are defined through axioms, which are invariant assertions about the histories of a particular transaction model. Unfortunately, the axiomatic definition might not be helpful in producing an operational transaction model and a corresponding transaction manager.

Similar to TSME, *Transaction Toolkits* enable the programmatic implementation of advanced transaction models by providing a set of transaction primitives. Using a C++ programming interface, *ASSET* [BDG⁺94] allows a transaction designer to compile programs with (advanced) transaction specification into executable code. *ASSET* is based on transaction primitives derived from the ACTA framework. Barga et al. have designed a *Reflexive Transaction Framework* [BP95, BP97] to implement different advanced transaction models. Using transaction adapters, i.e., add-on modules that are built on top of an existing transaction manager, they show how to extend the underlying functionality of the transaction manager.

Other interesting transactional workflow approaches are, among others, S-transactions [VEH92], the DOM transactions model [BOH⁺92], Polytransactions [SRK92], the combination of transactional workflows with semantics-based transaction management [BDS⁺93], IBM's Exotica project [MAGK95, AAA⁺95, AAA⁺96], the work of Günthör [Gün93, Gün96], the TAM transaction model [LP97, LP98], the WIDE two-level transaction model [CGS97,

GVBA97], and the work of Veijalainen et al. [PTV96, PTV97, TV95, VEH92] aiming at workflow support in inter-organizational environments (partially done in the context of the TRANS-COOP project).

The focus of all these transactional workflow approaches is on *coordination*, i.e., the coordinated execution of multiple related tasks, either performed by humans or automated tasks. The execution constraints are in most cases induced from data flow dependencies and organizational constraints.

Cooperation is characterized by passing results between (transactional) tasks in a predefined, programmed manner. Flexible but controlled sharing and exchange of data objects among co-workers participating in the same task or in different tasks (at the same or different times) is not in the focus of transactional workflow approaches. This is essential in non-workflow scenarios like cooperative authoring, Design for Manufacturing, and software development. Flexible revision of decisions (requirement (B)) and support for private and shared data (requirement (D)) are not addressed. However, the requirements of relaxed atomicity (A) and the coordination of work (E) are fulfilled quite well by transactional workflow models.

2.3.7 Comparison of Advanced Transaction Models

We conclude our discussion of advanced transaction models by comparing them with respect to support for cooperative applications. Table 2.1 summarizes to what degree the different categories of advanced transaction models fulfill the requirements presented in Section 2.2. A “⊙” in Table 2.1 indicates that the requirement is only partially fulfilled.

2.3.8 Other Approaches to Support Cooperation

Besides advanced transaction models, groupware systems and workflow management systems also support cooperation among multiple users. We conclude this chapter by discussing these approaches.

2.3.8.1 Groupware Systems

Most groupware systems [EGR91, Rod91, GP97] synchronize cooperative access to shared data in a more or less *ad-hoc* manner rather than using a well-defined formal notion of correctness for data sharing [VWP⁺98, KPTV95, TVF⁺95]. Concurrency control in *asynchronous* groupware systems is usually based on such mechanisms like explicit user-controlled locking of objects, different lock modes, extended lock semantics, and notifications [GS87, EGR91, WL93]. Some systems use floor passing protocols or turn-taking mechanisms [GS87] to synchronize concurrent operations on shared data, thereby heavily limiting the availability of data. Other systems do not provide any concurrency control at all and rely only on social protocols [EGR91], providing only visual feedback to indicate which objects are “in use”.

Table 2.1: Comparison of Advanced Transaction Models.

	(A) Relaxed Atomicity	(B) Retractions of Decisions	(C) Exchange of Work	(D) Private and Shared Data	(E) Coordination of Work	Comments
Semantics-Based Concurrency Control	—	—	—	—	—	Usage of operations semantics increases concurrency
Multi-Level and Open-Nested Transactions	⊙ ¹	⊙ ¹	—	—	—	Usage of level-specific operation semantics; ¹ Partial rollbacks by compensation
Split/Join Transaction Model, Delegation	⊙ ²	—	+ ³	—	—	² Adaptive recovery if direct commit after split; ³ Possible consistency problems
Check-Out/Check-In Models, Versioning etc.	⊙ ⁴	⊙ ⁴	⊙ ⁵	+	—	⁴ Immutable versions; ⁵ Requires manual merge, no formal correctness criteria
Group-Oriented Transaction Approaches	⊙	—	⊙	+	⊙	Properties depend heavily on the particular model (cf. Section 2.3.5)
Transactional Workflow Approaches	+	—	⊙	—	+	Properties depend heavily on the particular model (cf. Section 2.3.6)

The most prominent cooperative hypermedia authoring systems belonging to this category of asynchronous groupware are Notecards [Hal88, IT89], CoAuthor [JJKF89, EJ91], KMS [AMY88, YAM89], Intermedia [YHMD88, HKR⁺92], EHTS [Wii91, Wii92], HyperBase [WO90, Wii93], HyperForm [WL92], and CHS [SS90, SH93]. Quilt [LFK88, FKL88] and PREP [NKCM90, NKCM92, NKCM94] are examples for asynchronous group editors that use draft passing and annotations to enable cooperation among co-authors.

The major drawback of these systems is that they do not offer a generic mechanism to realize a broad range of cooperative applications. Most systems utilize ad-hoc mechanisms for sharing data which are often very restrictive and lack a formal notion of data consistency. Some systems offer hard-coded mechanisms for work coordination. Both recovery and retraction of decisions are generally not supported.

Other approaches in the CSCW area, e.g., [EG89, KP90, Ell92, GM94, SKSH96, RG96, SYZC97, SE98b] are only applicable to *real-time* cooperative work. This category includes synchronous group editors like GROVE [EG89, EGR89], ShrEdit [MO92], or DistEdit [KP90, KP93] and shared whiteboards such as GroupSketch [GB91], TeamPaint [IKG92], and WS-CRAWL [LHW92]. All of them are GUI-centered applications that follow the WYSIWIS

(“what you see is what I see”) paradigm [SBF⁺87]. They require that all co-workers are simultaneously present.

Most of these real-time groupware systems are based on data replication at each site and use multicast protocols like ISIS [BC91, BSS91] for synchronization purposes. This requires that all sites are permanently connected via the network. In general, real-time groupware systems do not address the issues of persistency of data and recovery to ensure fault-tolerant processing, nor do they support mobile and disconnected co-workers or cooperative work in high-latency, low-bandwidth networks. Since these systems are based on the WYSIWIS paradigm, they do not distinguish between private and shared data. Support for coordination of work and retraction of decisions is generally not available.

Most of the groupware systems mentioned in this section are research prototypes and are highly tailored to a specific application domain, e.g., cooperative authoring. On the other hand, in recent years some commercial groupware platforms have emerged that can be used to build cooperative applications (also classified as *meta-groupware* by CSCW researchers [EW94a]).

Two representative examples are Lotus Domino [Lot97, Lot96] (formerly known as Lotus Notes) and recent versions of Microsoft Exchange [Mic98]. These groupware platforms provide high-level communication support, message routing, document repositories, and tools to build special purpose applications that use the base functionality. By offering scripting languages, form-builders, etc. they allow an application designer to create an instance of a groupware system.

As such, workflow applications built on top of these groupware products become one of their future usage. For example, Domino provides the basic routing functionality to permit third-parties to build workflow-like applications with both database-based and mail-based architectures [Moh98]. Like in workflow management systems, features that allow the flexible sharing of documents in a cooperative fashion are missing in both groupware products, or are not very mature in the current versions. For example, Lotus allows the replication of database tables at record level but does not support system-supported conflict resolution mechanisms.

2.3.8.2 Workflow Management Systems

Unlike the field of groupware systems, the current state-of-the-art in workflow management is dictated by the commercial market [AS96, SK98]. Several hundred products that claim to provide support for building workflow applications exist in the market today [Moh98, GT98, ED98, CHRW98, SK98, AAAM97, GHS95].

Workflow management is gaining popularity, although the current generation of workflow management systems (WfMS) has severe limitations [GHS95, MAGK95, JLP⁺95, VPL95, AAAM97, Moh98, CHRW98, ED98, KWA99a, KWA99b]. Many of these deficiencies results from the strong process orientation of WfMS—neglecting data-centric issues, i.e., the management of application data. Deficiencies are, for example, the lack of support for correctness and data consistency (in the case of concurrent workflow tasks), and insufficient recovery mechanisms. The responsibility of data management is usually delegated by the WfMS to a com-

mercial database management system that holds the application data that is processed within a workflow application. Since the underlying database management system usually supports only traditional transactions, data consistency for workflow applications is difficult to achieve.

Research in the area of transactional workflows (cf. Section 2.3.6) provides appropriate directions to enhance WfMS in this respect and to better support relaxed atomicity [Ley95, Ley97]. For example, IBM discussed plans to integrate compensation mechanisms into their workflow product MQ Series Workflow [LR98], formerly known as FlowMark [LA94].

Another drawback of most commercial WfMS is that they concentrate on the coordination of relatively static workflows and on the modeling of the organizational aspects within workflow applications. Current WfMS do not support cooperative sharing of documents among workflow tasks. They do not offer adequate modeling and execution support for dynamically determined executions of activities, as required, e.g., in cooperative document authoring, multi-user software development, and Design for Manufacturing.

Recently, there has been an increasing interest in the research community in developing adaptive workflows and more dynamic WfMS. The majority of current work addresses relevant issues at modeling and language levels [EW94b, EKR95, Her95, HHSW96, LP97, HS98, CCPP98, LP98, Wes98, RD98], with some efforts in implementation underway [MGB⁺98, RHD98, RD98, HSS96, HHJ⁺99]. On the other hand, these approaches address the issue of flexible but consistent sharing and exchange of data among co-workers only marginally.

Some promising approaches towards integration of workflow coordination technologies with cooperation technology and groupware are investigated in [AG95, GP96, She97, MCC98, GAP98, SK98]. Directions for the support of mobile and disconnected users in workflow applications are discussed in [BMR96, AGK⁺96, CR98].

Chapter 3

The Cooperative Activity Model

This chapter describes the cooperative transaction model COACT and its underlying paradigm of cooperative work. As motivated in Section 2, cooperative applications impose new requirements on the transaction concept. In such environments, failure atomicity is too strict, and isolation among users contradicts the need of cooperation. A transaction model supporting co-operation should make the *semantically correct sharing and exchange of information* among co-workers as natural as possible, while still preserving consistency of the data processed.

The COACT model combines the use of private workspaces, that allow individual participants to work independently, with a synchronization mechanism called *history merging* that allows them to combine their work into a coherent whole. To this end, we take an *operation-oriented* view, rather than a state-oriented view: the consistency of common work results is determined based on the *semantics* of the operations performed. COACT exploits both backward commutativity *and* forward commutativity for this. The flexibility of our approach is mainly achieved by its ability to determine consistent units of work dynamically, and its consideration of operation semantics for detecting and resolving conflicts.

All cooperation facilities in COACT are based on the history merging paradigm. Merging histories with pre-specified commutativity relations provides transactional guarantees to a cooperative activity. In particular, we guarantee in COACT that all workspace histories remain *legal*, although information is exchanged. Using legality as its correctness criterion, history merging relaxes the isolation and atomicity properties of traditional transactions and enables cooperation among users—while ensuring consistency of their workspaces.

The chapter is structured as follows. First, we give an overview on the basic ideas behind the COACT model. In Section 3.2, we introduce a formal model of computation and the central concepts of operations, histories, and equivalence and legality of histories. Legality and equivalence of histories are used to define various kinds of commutativity relations in Section 3.3. In Section 3.4, we introduce the notion of cooperative activity types and cooperative activities. The nucleus of the cooperation facilities in COACT is the history merging approach, which is presented in Section 3.5. In Section 3.6, we describe the CoCoA coordination model (cf. Section 1.3) in more detail and assess the COACT cooperative transaction model with respect to the

requirements identified in Section 2.2. We conclude this chapter by comparing our approach with the advanced transaction models discussed in Section 2.3.

3.1 Overview of the Cooperative Activity Model

The *Cooperative Activity Model* (COACT, for short) [RKT⁺95a, RKT⁺95b, WK96, Wäs96, KTW96b, KTW96c, WAN98, KTWK98, KTW97b, KTW97a] starts from the observation that cooperative work is characterized by alternating periods of individual and joint work (cf. Section 2.1 and [Kai95, EMP⁺97, TVL⁺98]). During individual work periods, users explore alternative problem solutions, while co-workers may work simultaneously on the same subject. Access to and use of shared data objects should neither block other users, nor should it affect co-workers unintendedly. During joint work, co-workers should be able to exchange information and to share final as well as intermediate results. Co-workers should be able to form subgroups dynamically. Joining and leaving an ongoing cooperative activity should be possible.

Therefore, we assign a *private workspace* to every user who participates in a collaborative effort, i.e., a *cooperative activity*. By default, the private workspaces of the co-workers are isolated from each other and can only be accessed and manipulated by the workspace owner. Additionally, there exists a *common workspace* per cooperative activity.

The common workspace is isolated from the private workspaces and is not assigned to a single actor in the cooperative activity. The common workspace contains the data objects available when a cooperative activity is *started*, and it will contain the results of the cooperative activity when it is *committed*. Figure 3.1 gives an illustration of these constituents of the COACT model.

To achieve isolation of workspaces, we (conceptually) copy the data objects initially contained in the common workspace to all private workspaces (initial state of the cooperative activity). These copies of the objects can then be manipulated independently in the private workspaces. Hence, the modifications to data objects done by different co-workers do not interfere. This conceptual design is well-suited to support cooperation among geographically distributed users or mobile users that are only partially connected to the network. In such environments, it is too strict to require that all co-workers are permanently connected with each other.

We assume in COACT an environment, where the sequence of operations executed in a workspace is composed interactively by the user. Each user interaction with the corresponding workspace is modeled as a single atomic operation. A user selects the next operation from a predefined set of *operation types* (see next page) and provides the actual input parameter values to that operation. The invocation of the predefined operation types is the only means to retrieve and change the state of the own private workspace. The resulting *operation instances* (operations, for short) appear as atomic changes on the respective workspace that transfer a consistent workspace state into another consistent workspace state.

For each workspace, we record all modifications in a *workspace history*. A workspace

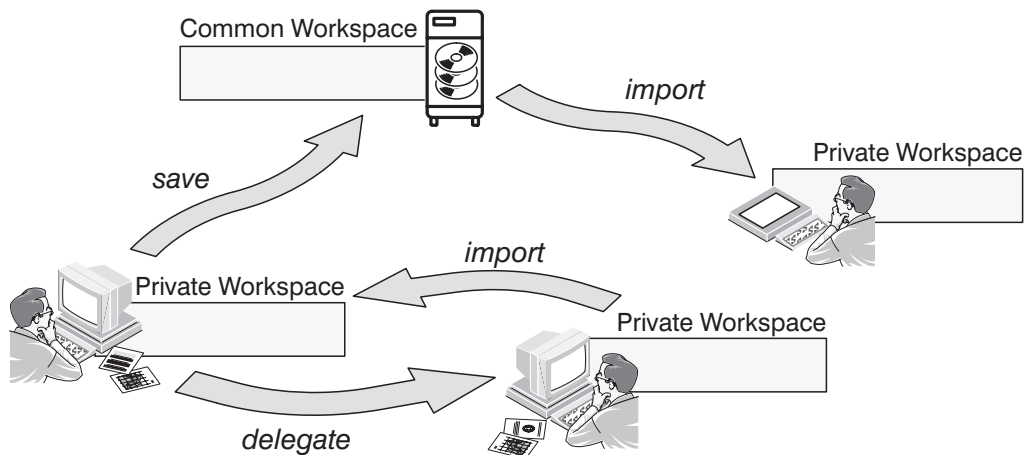


Figure 3.1: Workspaces and Exchange Facilities in the COACT Model.

history is represented as a logical log of operations executed in a workspace. Each history entry contains all information about the respective operation execution, including the input and output parameter values of the operation. Using write-ahead logging of operations, the workspaces within a cooperative activity become durable.

To model the correct interaction, i.e., cooperation of users involved in a cooperative activity, we introduce the concept of *cooperative activity types* (cf. Section 3.4.1).

- First, a cooperative activity type describes a set O of *invocable operation types* that can be invoked by a user, i.e., generated, in his private workspace.
- Second, a cooperative activity type specifies for each operation type $ot \in O$ a *compensation function*. The compensation function is used to compute a *compensation operation* that *semantically undoes* the effects of the original operation.
- Third, a cooperative activity type specifies type-specific merging relations M , described by *commutativity relations* among operations. The commutativity relations guide the process of an information exchange (history merging).

Exchange of information among workspaces is an explicit act, that is initiated by an actor through invoking one of COACT's *exchange meta-operations*, namely *import*, *delegate*, and *save* (cf. Figure 3.1). The exchange meta-operations are generically provided by the COACT model, like starting, aborting, or committing of a cooperative activity¹.

All of them are based on the history merging mechanism, which exchanges operations among workspaces, rather than the data objects themselves. Essentially, the merging of histories can be viewed as incorporating a subsequence of operations from a source workspace

¹The name “meta-operations” results from the fact that exchange meta-operations operate on sequences of operations that manipulate the data objects in the workspaces.

history into a target workspace history. History merging ensures that the source and target workspaces always keep consistent. Conflicts that may occur during an exchange meta-operation are resolved by undoing operations in the target workspace and/or by neglecting some of the operations to be incorporated.

The COACT cooperative transaction model provides the following options for exchanging information among workspaces:

1. Co-workers can directly exchange partial results between their private workspaces by means of the *import* and *delegate* exchange meta-operations.
 - *Import* is used by a co-worker to incorporate a set of operations executed in the scope of another workspace into the own workspace. The importing user is responsible for resolving conflicts that may occur during the merge. An *import* has no effect on the workspace from which the work is being imported.
 - *Delegate* is used to pass on a set of operations to a co-worker (the delegatee), who is then responsible for merging the operations into his private workspace. Conflicts that may occur are resolved under the supervision of the delegatee. The *delegate* exchange meta-operation has no effect on the workspace of the delegator.
- The difference is that, in the former case, the actor who initiates the exchange meta-operation is also responsible to control the merge with his private workspace, whereas in the latter case, the delegator only selects the operations and the delegatee controls the merge with his workspace.
2. Co-workers can exchange information through the common workspace by means of the *save* and *import* exchange meta-operations.
 - A user can use *save* to incorporate operations of his private workspace into the common workspace, thus, making (parts of) his results public to all co-workers. The user who invokes a *save* is responsible for the resolution of possible conflicts in the common workspace. Only the common workspace is modified by a *save*—the private workspace of the user remains unchanged.
 - Other co-workers can then retrieve and incorporate these (and other) operations from the common workspace into their private workspaces, using the *import* exchange meta-operation described above. The resolution of possible conflicts is done under the supervision of the importing user. The common workspace remains unchanged.

In contrast to a *delegate*, the user that initiates the *save* is also responsible for resolving possible conflicts in the target workspace (in this case the common workspace).

The history merge mechanism allows only to exchange consistent parts of workspaces. A consistent part of a workspace is represented as a *consistent unit of work*, which is a subsequence of the operations in a workspace history. We identify consistent units of work by examining the *backward commutativity relation* [Wei88a, LMWF94] between operations contained in a workspace history. A user selects (e.g., via a graphical user interface) the operations to be exchanged, and COACT *dynamically* computes the minimal consistent sequence of operations to be exchanged.

The incorporation of these operations into another workspace is then realized by the *re-execution of operations* in the respective target workspace. In this way, the effects of the operations are reflected in the state of the target workspace.

The semantic correctness of the exchange of operations is guaranteed by ensuring that the re-execution of an operation has an equivalent “view” on the history, in both the target workspace and the source workspace. Hence, the behavior of a re-executed operation in terms of output parameter values is indistinguishable from its initial execution, i.e., the re-executed operation remains legal. We use the *forward commutativity relation* [Wei88a, LMWF94] to check this.

If the merge process cannot be performed without violating the semantical correctness of the target workspace history, the merge mechanism offers different alternatives to proceed, i.e., it identifies different consistent sets of operation sequences. In this case, the controlling user can select one of those consistent operation sequences to be incorporated in a workspace.

To ease the merge process for the user, the task of selecting an alternative for the merge can be supported by an extra decision module, without requiring user interactions. In this case, certain merge policies can be specified providing different conflict resolution strategies.

The resolution of conflicts during a merge may result in retracting previous decisions, i.e., undoing operation sequences in the target workspace. This is accomplished by means of *compensation* in COACT. Using the compensation functions specified and the backward commutativity relation in M , we ensure that only the minimal amount of work is discarded. Additionally, our compensation approach allows users to *selectively undo* operations in their private workspaces, which is important for the interactive processing of activities.

If an operation has been successfully incorporated into another workspace, it is conceptually the *identical* operation as in the original workspace. The *sharing* of operations among several workspace histories allows us to achieve a close cooperation between co-workers, because certain conflicts with identical operations that are present in both workspace histories can be ignored during a merge.

The degree of cooperation that can be achieved by history merging depends on the exchange frequency. The more often operation sequences are exchanged among workspaces, the more identical operations the workspaces will contain. If in all workspaces the same (automatic) conflict resolution policy is applied, the workspace states within a cooperative activity will eventually converge, i.e., the states of all workspaces will be equivalent.

Besides the mentioned meta-operations for the exchange of information among workspaces,

the COACT model provides basic primitives to manage the execution of cooperative activities (cf. Section 3.4.3). These primitives include the instantiation of a cooperative activity of a given type (*start*), the ability to *suspend* a user's activity to be resumed later on, and primitives to *join* or *leave* the cooperative activity. *Join* creates a new workspace for a user in an already existing cooperative activity, while *leave* discards the workspace of the corresponding user who is no longer part of the cooperative activity. Before leaving, users will delegate their results to co-workers or save them in the common workspace.

In addition, the COACT model provides primitives for aborting or committing a cooperative activity with almost usual semantics. Those parts of a cooperative activity that are reflected in the common workspace after its commit are considered as the final result of the cooperative activity. It is assumed that all users integrate their relevant contributions into the common workspace, such that there is a single result of the cooperative activity. Commit and abort discard all the private workspaces. In case of an abort, also the common workspace is discarded.

Note, that in principle, the *import* and *delegate* exchange meta-operations among private workspaces could be eliminated, requiring all users to cooperate only through the common workspace. We decided to provide the option to exchange information directly among private workspace, since it allows subsets of users to cooperate, which would be impossible otherwise. On the other hand, information exchange via the common workspace (by means of *save* and *import*) can be eliminated, requiring all users to exchange information directly among their private workspaces. We decided to provide this exchange option via the common workspace, since it allows a user to make (parts of) the information in his private workspace public to *all* co-workers. Additionally, we defined the common workspace to contain the initial data set of a cooperative activity when it starts and the result of the cooperative activity when it finishes. For this, *import* and *save* between the common and the private workspaces are needed.

3.2 Model of Computation

In this section, we define the basic building blocks of the computational model that allow us to define different kinds of commutativity relations, the history merging paradigm, and COACT's notion of correctness.

In our data model there exist two kinds of entities: data objects and operations. Objects encapsulate the data items contained in a workspace. Operations are the only means to examine and modify the state of a workspace, i.e., the state of the data objects contained in a workspace. We further distinguish between operation types and operation instances. The former describe the semantics of state transitions within a workspace. Operations are individual instantiations of operation types. Operations record their input parameters values together with their results. This allows us to define when an operation (re-)execution is considered as legal. Legality of operations is a central concept in COACT. It describes if the invocation and response events of an operation match for a certain workspace state.

Complex computations are modeled in COACT as finite sequences of operations, called

histories. We consider a history legal if all of its operations are legal. The notion of legal histories allows us to derive the concepts of equivalence of workspace states and equivalence of histories. Within this framework, we are able to formally describe the commutativity relations used in CoACT (cf. Section 3.3) and CoACT's history merging mechanism (cf. Section 3.5).

3.2.1 Data Model

In the following, we introduce CoACT's underlying abstract data model. It provides the central concepts needed for the definition of the CoACT cooperative transaction model, omitting unessential details (in particular data structures).

3.2.1.1 Objects and Workspaces

The basic containers for data are called *data objects* (or objects, for short). Each data object has a *value*, a *domain* and a *unique object identifier* (OID) assigned to it. The domain of a data object defines the possible set of values the object can have. The OID of an object and its domain are *immutable*, whereas the value of a data object can change over time.

DEFINITION 1 (DATA OBJECTS)

A data object d is a triple $d = (i, v, V)$ where i is the object identifier, V is the domain of the object, and $v \in V \cup \{\perp\}$ is the current value of the object d . If $v = \perp$, then the value of the object is undefined. With $oid(d)$ we denote the object identifier i , $value(d)$ denotes the current value v , and $dom(d)$ denotes the domain of object d .

Object identifiers are needed to identify and distinguish data objects, because different objects can have the same values. Two objects are considered to be *identical* only if they have the same object identifier. Two objects are considered *equal* if their values are the same.

For the sake of simplicity, we assume that there exists an infinite set of object identifiers and that an object identifier is used only once. In practice, object identifiers are system-generated values and the underlying computing system ensures that object identifiers are *system-wide unique*.

DEFINITION 2 (IDENTITY OF DATA OBJECTS)

Two data objects d and e are *identical* ($d == e$) iff $oid(d) = oid(e)$.

A *workspace* (or database) is a collection of data objects. All objects contained in the workspace must be assigned a value out of their domain, i.e., a data object d exists in a workspace only if $value(d) \neq \perp$. If the value of an object is $\perp$, this means that the object is not yet “created” or that the object is already “deleted”.

DEFINITION 3 (WORKSPACE)

A *workspace* (or database) WS is a set $\{d_1, d_2, \dots, d_n\}$ of data objects with $\neg (d_i == d_j)$, for all $i \neq j$, and with $value(d_i) \neq \perp$, for all $i = 1, \dots, n$.

Because the values of data objects in a workspace can change over time, we introduce the concept of a workspace *state*. A state describes which objects are contained in a workspace at a particular point of time and which values these data objects have. Two workspace states are *equal* if the data objects have the same values.

DEFINITION 4 (STATE)

Let D be the set of all data objects that can occur in a workspace. Then, a *state* S is a mapping that assigns to each $d \in D$ a corresponding value from $\text{dom}(d) \cup \{\perp\}$, i.e., $S(d) = v$ where $v \in \text{dom}(d) \cup \{\perp\}$. If S denotes the state of a workspace WS , we define $d \in WS$ iff $S(d) \neq \perp$. With $S_\perp$ we denote the empty state, i.e., $\forall d \in D : S(d) = \perp$. With $\mathbb{S}$ we denote the set of all possible states of a workspace.

DEFINITION 5 (EQUALITY OF STATES)

Let D be the set of all data objects that can occur in a workspace. Two states S and T are *equal* ($S = T$) iff $\forall d \in D : S(d) = T(d)$.

As a shorthand, we denote by $S[d \mapsto v]$ the state obtained by mapping object d to value v and leaving the mapping S otherwise unchanged.

In the following, we present a bank account example to illustrate the concepts introduced so far. It is based on [Wei88a, Wei89a] and rather simple, thus easy to understand, but sufficient to illustrate all the concepts we introduce in this section. Variants of this example are widely used in the field of semantics-based transaction processing, e.g., [Her86, Wei88a, Wei89a, Her90, AAS93, AA94, Mut94, Rak97].

Example 11 (Bank accounts) Each bank account in the ACCOUNTS database is modeled as a data object whose value represents the name of the account owner (as a string) and the account balance (as a real number), i.e., $V = \text{STRING} \times \text{REAL}$.

Assume that ACCOUNTS contains three account objects a, b, c . Let state S_1 be $S_1(a) = (\text{Alice}, 1000)$, $S_1(b) = (\text{Bob}, 900)$, and $S_1(c) = (\text{Alice}, 900)$. In this state, all objects $d \in \text{ACCOUNTS}$ are not equal. Note that there exist two account objects for Alice which are only distinguished by their object identifiers. If Bob transfers 100 DM from his account b to account c , this yields the state S_2 with $S_2(a) = (\text{Alice}, 1000)$, $S_2(b) = (\text{Bob}, 800)$, and $S_2(c) = (\text{Alice}, 1000)$. Now, a and c are equal but not identical, since they have different object identifiers. Of course, $S_1 \neq S_2$. Assume, Alice transfers from her account a 100 DM to Bob's account b and then 100 DM from account c to account a . These money transfers result in state S_3 with $S_3(a) = (\text{Alice}, 1000)$, $S_3(b) = (\text{Bob}, 900)$, and $S_3(c) = (\text{Alice}, 900)$. Thus, state $S_3 = S_1$. $\square$

In object-oriented data models, objects are usually further classified into *object types*. An object type T describes the internal structure for a set of homogeneous data objects. A total typing function $\text{type} : D \rightarrow \mathbb{T}$ assigns to each object $d \in D$ an object type $T \in \mathbb{T}$. For example, we can introduce an object type ACCOUNTOBJECT with two attributes *name* and *amount*

where $\text{dom}(\text{name}) = \text{STRING}$ and $\text{dom}(\text{amount}) = \text{REAL}$. This object type describes the account objects contained in the ACCOUNTS database.

Such explicit typing is used in most object-oriented database models like VML [KAN94, DIM95] or ODMG [Cat94] or object-oriented programming languages like Java [AG98]. In addition to the structure of objects, object types in object-oriented models also describe the *behavior* of objects through *methods* or operation types assigned to the object types. In contrast to this, we use a more generic model in COACT. Operation types are kept separate from the objects. This is similar to SQL:1999 (formerly known as SQL-3) user-defined routines which are not assigned to a type or table in the database [EM99, Wäs97, IA97, Mel98].

Therefore, and since explicit typing would make our model unnecessarily complex, we do not use object types in the following. Also, we do not introduce extensional classes nor do we model inheritance. All of these concepts are not essential for COACT (but could easily be introduced into our model).

3.2.1.2 Operation Types

To describe *transitions* among workspace states, we introduce the concept of *operation types*. An operation type can be compared to the definition of a procedure or method in a programming language. An operation type with name ot has formal input parameters $in_1, \dots, in_n$ with domains $\text{dom}(in_1), \dots, \text{dom}(in_n)$ and formal output parameters $out_1, \dots, out_m$ with domains $\text{dom}(out_1), \dots, \text{dom}(out_m)$, where n is the number of input parameters and m is the number of output parameters. In the following, we refer to the *signature* of an operation type as $ot(\text{dom}(in_1) in_1, \dots, \text{dom}(in_n) in_n) : [\text{dom}(out_1) out_1, \dots, \text{dom}(out_m) out_m]$.

Note that the state of objects can only be observed and modified through the invocation of operation types. This means that a user can only observe that part of a workspace that is *externalized* by the output parameters of operation types.

In COACT, we abstract from the concrete implementation of an operation type by using functions to specify the semantics of operation types. An operation type is described by a deterministic function f_{ot} . The input of f_{ot} is a workspace state $S \in \mathbb{S}$ and the input parameter values in_i . The result of the function f_{ot} are the output values out_j and the new state $T \in \mathbb{S}$. Depending on the actual state S and the input parameters, the application of f_{ot} transforms the state S into T , i.e., it alters the values of objects in the workspace.

DEFINITION 6 (OPERATION TYPE)

An operation type ot with input parameters $in_1, \dots, in_n$ and output parameters $out_1, \dots, out_m$ is a deterministic function $f_{ot} : \mathbb{S} \times I \rightarrow \mathbb{S} \times O$ with $I = \text{dom}(in_1) \times \dots \times \text{dom}(in_n)$, $O = \text{dom}(out_1) \times \dots \times \text{dom}(out_m)$.

Note that the function f_{ot} has to be *deterministic*. This means that an operation type cannot return random values or make random state transitions. This property is important in order to re-execute operations in other workspaces. We require that f_{ot} is *total*, i.e., for every combination

of input state and input parameters there is a defined output state and defined result values². For the sake of simplicity, we assume that an operation type is always invoked with a valid combination of input parameters values from the respective parameter domains.

Example 12 (Operation types for ACCOUNTS) Consider the ACCOUNTS example introduced in the last section. The available operation types for ACCOUNTS are:

***createAccount*(STRING *name*):[OBJECT *account*]** takes a string as input and returns a newly “created” account data object *account* with $value(account) = (name, 0)$. With OBJECT we denote the set D of all possible data objects (cf. Definition 4). The semantics of *createAccount* is described by the following function:

$$\begin{aligned} createAccount &: \mathbb{S} \times \text{STRING} \rightarrow \mathbb{S} \times \text{OBJECT} \text{ with} \\ createAccount(S, name) &= (T, account) \end{aligned}$$

where $S(account) = \perp$, $T = S[account \mapsto (name, 0)]$. Recall that $S[d \mapsto v]$ denotes the state that is identical to S , except that object d is mapped to value v .

***credit*(OBJECT *account*, REAL⁺ *amount*)** takes an existing *account* object and a non-negative *amount* as input and increments the balance of *account* by *amount*—other objects remain unchanged:

$$\begin{aligned} credit &: \mathbb{S} \times \text{OBJECT} \times \text{REAL}^+ \rightarrow \mathbb{S} \text{ with} \\ credit(S, account, amount) &= T \end{aligned}$$

where $S(account) = (name, balance)$, $T = S[account \mapsto (name, balance + amount)]$. The specified operation type has no externally visible effects, i.e., it has no output parameters.

***debit*(OBJECT *account*, REAL⁺ *amount*)** decrements the balance of *account* by the specified *amount*:

$$\begin{aligned} debit &: \mathbb{S} \times \text{OBJECT} \times \text{REAL}^+ \rightarrow \mathbb{S} \text{ with} \\ debit(S, account, amount) &= S' \end{aligned}$$

where $S(account) = (name, balance)$, $T = S[account \mapsto (name, balance \Leftarrow amount)]$.

***owner*(OBJECT *account*):[STRING *name*]** returns the name of the account owner:

$$\begin{aligned} owner &: \mathbb{S} \times \text{OBJECT} \rightarrow \mathbb{S} \times \text{STRING} \text{ with} \\ owner(S, account) &= (S, \Pi_1(S(account))) \end{aligned}$$

where $\Pi_1(S(account))$ denotes the projection onto the first component of $S(account)$. Note, *owner* does not change the state of an object—it only has an external effect, i.e., the name of the account owner is returned.

²This is no severe restriction, since we can transform every partial function into a total one that has a resulting state equal to the input state and “*nil*” output values in case the result of the partial function is undefined. This only requires that the domains of the output parameters for the total function include an additional “*nil*” value.

balance(**OBJECT** *account*):[**REAL** *amount*] returns the current balance of *account*:

$$\begin{aligned} \text{balance} &: \mathbb{S} \times \text{OBJECT} \rightarrow \mathbb{S} \times \text{REAL} \text{ with} \\ \text{balance}(S, \text{account}) &= (S, \Pi_2(S(\text{account}))) \end{aligned}$$

transfer(**OBJECT** *account*₁, **OBJECT** *account*₂, **REAL**⁺ *amount*) transfers the specified *amount* of money from *account*₁ to *account*₂ in a *single* atomic state transition:

$$\begin{aligned} \text{transfer} &: \mathbb{S} \times \text{OBJECT} \times \text{OBJECT} \times \text{REAL}^+ \rightarrow \mathbb{S} \text{ with} \\ \text{transfer}(S, \text{account}_1, \text{account}_2, \text{amount}) &= T \end{aligned}$$

where $S(\text{account}_1) = (\text{name}_1, \text{balance}_1)$, $S(\text{account}_2) = (\text{name}_2, \text{balance}_2)$, and state $T = S[\text{account}_1 \mapsto (\text{name}_1, \text{balance}_1 \ominus \text{amount})][\text{account}_2 \mapsto (\text{name}_2, \text{balance}_2 + \text{amount})]$.

total(**STRING** *owner*):[**REAL** *total*] returns the sum of the balances of all accounts belonging to the specified *owner*:

$$\begin{aligned} \text{total} &: \mathbb{S} \times \text{STRING} \rightarrow \mathbb{S} \times \text{REAL} \text{ with} \\ \text{total}(S, \text{owner}) &= (S, \sum_{\Pi_1(S(d))=\text{owner}} \Pi_2(S(d))) \end{aligned}$$

deleteAccount(**OBJECT** *owner*) takes a data object as input and “deletes” it:

$$\begin{aligned} \text{deleteAccount} &: \mathbb{S} \times \text{OBJECT} \rightarrow \mathbb{S} \text{ with} \\ \text{deleteAccount}(S, \text{account}) &= S[\text{account} \mapsto \perp] \end{aligned}$$

□

Objects and operation types are orthogonal concepts in CoACT. The main reason for not associating objects and operation types in CoACT is that we do *not* want to restrict the effects of an operation to a *single* data object. Most other semantics-based transaction models like those found in [GM83, SS84, Her86, BHG87, HW88, Wei88a, Her90, Wei93, AAS93, AA94] assume that an operation can only observe or modify the state of a single object. This implies that it is *not* possible in these transaction models to describe operation types with richer semantics, e.g., a *transfer* that affects the state of two data objects. More complex operations have to be explicitly modeled as transaction programs then. For example, *transfer* can not be modeled as an operation type—it has to be a transaction in those approaches. This is not necessary in CoACT, because an operation can be arbitrarily complex. A drawback is that the specification of commutativity relations might become more complex, since we have to (theoretically) specify commutativity between all operation types *ot* that can be invoked by a user, i.e., $ot \in O$. The reason is that the local atomicity property [Wei89b, Wei88a, Her90, LMWF94] can no longer be guaranteed since the invocation of an operation can potentially observe or change the state of all objects in a workspace.

3.2.2 Operations

Operation types only describe formal parameters. If an operation is applied, the formal input parameters are replaced by actual input parameter *values*. In CoACT, we denote the appli-

cation of an operation to a particular state with actual input parameter values as an *operation invocation*.

DEFINITION 7 (OPERATION INVOCATION)

An *operation invocation* i is modeled as a tuple $i = (ot, \vec{in})$. The first component of i is the operation type ot ; $\vec{in} = (in_1, \dots, in_n) \in I$ are the actual input parameter values. By $type(i)$ we denote the operation type ot of i , and $input(i)$ denotes the input parameter values $\vec{in}$ of i . By Σ we denote the (not necessarily finite) set of operation invocations.

For the sake of brevity, we write in the following $ot(\vec{in})$ to denote an operation invocation $i = (ot, \vec{in})$. Note that in operation invocations only the input parameter values are included but not the actual database state to which the operation type was applied. To denote the result of an operation invocation $ot(\vec{in})$ in state S we write $ot(\vec{in})(S)$, where $ot(\vec{in})(S) = f_{ot}(S, \vec{in})$.

Example 13 (Operation invocations) In the following, we give an example for operation invocations. Consider the database state S_1 in Example 11: $S_1(a) = (Alice, 1000)$, $S_1(b) = (Bob, 900)$, and $S_1(c) = (Alice, 900)$. Assume, *total* is invoked with input parameter *Alice*. This operation invocation returns 1900 as the total amount, i.e., $total(Alice)(S_1) = (S_2, 1900)$ with $S_1 = S_2$. A subsequent operation invocation $transfer(a, c, 100)$ yields state S_3 with $S_3(a) = (Alice, 900)$, $S_3(b) = (Bob, 900)$, and $S_3(c) = (Alice, 1000)$. $\square$

Operation invocations model only the *invocation events* of an operation type but not the *response events*, i.e., the actual output parameter values. To capture also the actual output parameter values, we introduce the concept of *operations*. An operation is a single instantiation of an operation type. It records the operation type invoked together with the actual input parameter values. Additionally, the output parameter values of the operation invocation are recorded in the operation. We assume in COACT that an operation is *atomic*, i.e., it is guaranteed that the operation invocation terminates successfully or it does not occur at all.

So far, two operations with the same input parameter values and the same output parameter values can not be distinguished. For instance, assume the operation type *total* is invoked with input parameter value *owner* = *Alice* in state S_1 and state S_3 of Example 12 on page 50. Both operations yield the same output value, i.e., 1900, and cannot be distinguished.

Identity of operations is a crucial concept in the COACT model as we will see later. To distinguish operations with the same input and output values, we introduce *operation identifiers*. Each operation gets assigned a *system-wide unique* identifier during its generation (cf. Definition 11). This identifier enables us to identify *identical* operations that may occur in different workspaces. This leads us to the following definitions.

DEFINITION 8 (OPERATION)

An *operation* o is modeled as a tuple $o = (ot, \vec{in}, \vec{out}, id)$. The first component of o is the operation type ot ; $\vec{in} = (in_1, \dots, in_n) \in I$ are the actual input parameter values of the operation invocation, and $\vec{out} = (out_1, \dots, out_m) \in O$ are the actual output parameter values. The last component id represents the operation identifier. By $type(o)$ we denote the operation type ot

of o , $input(o)$ and $output(o)$ denote the input parameter values $\vec{in}$ and the output parameter values $\vec{out}$, and $id(o)$ denotes the operation identifier. With Ω we denote the (not necessarily finite) set of all possible operations.

DEFINITION 9 (INSTANCES OF OPERATION INVOCATIONS)

An operation $o \in \Omega$ is an *instance* of an operation invocation $i \in \Sigma$ iff $type(o) = type(i) \wedge input(o) = input(i)$.

DEFINITION 10 (IDENTITY OF OPERATIONS)

Two operations $o, p \in \Omega$ are *identical* ($o = p$) iff $id(o) = id(p)$.

For brevity, we also write in the following $ot_{id}(in_1, \dots, in_n):[out_1, \dots, out_m]$ instead of $(ot, \vec{in}, \vec{out}, id)$ and $ot_{id}(in_1, \dots, in_n):out_1$ if ot has a single result value. For instance, we denote by $total_1(Alice):1900$ the operation $o = (total, Alice, 1900, 1)$.

Example 14 (Operations) Reconsider the operation invocations in Example 13. The computation there can simply be described as a sequence of operations starting from state S_1 . The sequence is: $total_1(Alice):1900$, $transfer_2(a, c, 100)$. $\square$

In COACT, the exchange of information among workspaces is realized by the semantically correct exchange of operations among the workspaces. History merging installs the effects of the exchanged operations by re-executing them in the target workspace (cf. Section 3.1). Since operations can be “shared” among workspaces, we need to distinguish the original execution, i.e., the *generation* of an operation, from its *re-execution*. This is captured in the following definition. Note that a re-executed operation o' gets assigned the same operation identifier as its original operation o , i.e., $id(o) = id(o')$, and, hence, o and o' are identical operations.

DEFINITION 11 (GENERATION AND RE-EXECUTION OF OPERATIONS)

By the *generation* (or *original execution*) of an operation $o = ot(\vec{in}):\vec{out}$ we denote the operation resulting from the invocation of $ot(\vec{in})$ on behalf of a user in his private workspace.

By the *re-execution* of an operation $o = ot(\vec{in}):\vec{out}$ we denote the operation o' resulting from the invocation of $ot(\vec{in})$ on behalf of the merger during an exchange meta-operation. The re-executed operation o' gets assigned the same operation identifier as operation o , and hence $o = o'$ holds.

Operations describe only the externally visible effects of operation invocations, i.e., the actual input and output parameter values, but neither the states before the operation nor the state after the operations. The output parameter values of operations provide only a limited knowledge on the workspace state (recall that we do not have direct access to the workspace). Because Definition 8 does not take into account the state S before an operation $o = ot(\vec{in}):\vec{out}$, it can be the case that $output(ot(\vec{in})(S)) \neq \vec{out}$, for example, when the operation o is re-executed in a different workspace state S' . Therefore, we define next whether invocation and response events *match* for a certain state.

DEFINITION 12 (LEGAL OPERATION)

An operation $o = (ot, \vec{in}, \vec{out}, id)$ is *legal* in a state $S \in \mathbb{S}$ iff the operation invocation $ot(S, \vec{in})$ yields output values $\vec{out}$, i.e., $\Pi_{2,\dots,m+1}(ot(S, \vec{in})) = \vec{out}$. $\Pi_{2,\dots,m+1}$ denotes the projection of the function result to the m last components, i.e., the output parameter values.

Legality of operations is the central concept in the COACT cooperative transaction model. We use this concept in the following sections to define equivalence of states, commutativity of operations, consistent units of work, and the correctness of information exchange in COACT.

A legal operation can only be re-executed in another user's workspace if it exhibits the same behavior as in its original execution (cf. Section 3.1). This implies that an operation can only be re-executed, if we can ensure that it remains legal. The history merging mechanism ensures that all operations exchanged among workspaces remain legal. In the following, we give examples of legal and illegal operations.

Example 15 (Legal and illegal operations) Assume that the state S_1 of ACCOUNTS is $S_1(a) = (Alice, 1000)$, $S_1(b) = (Bob, 900)$, and $S_1(c) = (Alice, 900)$. In state S_1 the operation $o_1 = total_1(Alice):1900$ is legal. The resulting state S_2 is equal to state S_1 . Now, operation $o_2 = transfer_2(a, b, 100)$ transfers 100 DM from Alice's account a to Bob's account b . This yields state S_3 with $S_3(a) = (Alice, 900)$, $S_3(b) = (Bob, 1000)$, and $S_3(c) = (Alice, 900)$. Operation o_2 is legal since it has no output parameters. Operation o_1 is *not* legal in state S_3 , e.g., when re-executed, since it does not return the correct total balance of Alice's account objects, which is 1800. $\square$

Most transaction models do not consider the return values of operations [Rak97], since they use pessimistic concurrency control policies that block operation invocations in case of conflicting access to data objects. Obviously, no return values are available for blocked operations. An exception is [Rak97] where the return values of operations that hold a lock are considered. In some optimistic concurrency control schemes, the return values are used, too, because operation results are easily available there [Her86, HW88, Her90, LMWF94].

In COACT operations are executed in a single-user workspace, where they can never be blocked. This enables us to use the return values of operation invocations in the definition of our cooperative transaction model. Using the return values, we can deduce some information about the states before and after an operation execution. This allows us to exploit the operation semantics at a more fine-grained level in the definition of commutativity relations.

3.2.3 Histories

To model complex computations in COACT, we introduce the concept of *histories*. A history is a finite sequence of operations that is totally ordered. We do not use a partial order since in COACT operations are executed interactively by a user in his private workspace. This implies that a user executes one operation after the other within a single thread of control. This is compliant with the policies used in most hypermedia and text editors, word processors (like

Microsoft Word), graphic tools, and programming environments. For the sake of simplicity, we do not consider intra-workspace parallelism since its implementation is orthogonal to the concepts of COACT³.

DEFINITION 13 (HISTORY)

A history $H = (O, <_O)$ is a finite set $O \subseteq \Omega$ of operations together with a total order $<_O \subseteq \Omega \times \Omega$ over the operations $o \in O$, reflecting the execution order of operations.

In the following, we introduce some shorthand notations that are useful for the subsequent presentation of COACT. We use capital letters to denote histories and small letters to denote operations. Instead of $H = (O, <_O)$, we often write $[o_1, \dots, o_n]$ to refer to a history H . We often use the notations $o \in H$ instead of $o \in O$ and $o <_H p$ instead of $o <_O p$ when the meaning is clear; $op(H)$ is used to refer to the set of operations O of a history $H = (O, <_O)$. By $H \bullet I$ we denote the *concatenation* of two histories $H = [o_1, \dots, o_n]$ and $I = [p_1, \dots, p_m]$, i.e., $H \bullet I = [o_1, \dots, o_n, p_1, \dots, p_m]$. If o is an operation then $H \bullet o$ stands for $H \bullet [o]$.

Histories usually change the state of workspaces. The resulting state of a history is dependent on the state S_0 before the first operation in a history. With $S_n = [o_1, \dots, o_n](S_0)$ we refer to the state S_n resulting from the execution of the history $[o_1, \dots, o_n]$ in state S_0 , i.e., $S_i = \Pi_1(type(o_i)(S_{i-1}, input(o_i)))$ where S_{i-1} is the state resulting from the execution of o_{i-1} , for $i = 1, \dots, n$.

Next, we define *equality* of histories. Two histories are *equal* if both histories contain exactly the same operations in the same order.

DEFINITION 14 (EQUALITY OF HISTORIES)

Two history $H_1 = (O_1, <_{O_1})$, $H_2 = (O_2, <_{O_2})$ are *equal* ($H_1 = H_2$) iff $O_1 = O_2 \wedge <_{O_1} = <_{O_2}$.

In the description of the COACT model, we often refer to parts of histories. A *subhistory* is a subsequence of a history with the restriction that the ordering of operations must be preserved. The *prefix* of a history is defined as usual.

DEFINITION 15 (SUBHISTORY AND RESTRICTION OF A HISTORY)

A history $I = (P, <_P)$ is called a *subhistory* of $H = (O, <_O)$ iff $P \subseteq O \wedge \forall o, p \in P : o <_O p \Leftrightarrow o <_P p$. With the *restriction* of $H = (O, <_O)$ to operations $P \subseteq O$ we refer to the subhistory $I = (P, <_P)$ of H .

DEFINITION 16 (PREFIX, CONTINUATION, AND REORDERING OF A HISTORY)

A history I is called a *prefix* of H iff there exists a history J with $H = I \bullet J$. A history I is called a *continuation* of H iff H is a prefix of I . A history I is a *reordering* of H iff $op(I) = op(H)$.

³However, the restriction to a total order does not preclude intra-workspace parallelism. Isolation of concurrently executing operations can be achieved by a simple (low-level) intra-workspace concurrency control protocol. Concurrently executing operations can be totally ordered by their invocation or response event (depending on the concurrency control policies used).

Example 16 (Histories, subhistories, prefixes, and continuations) The computation in Example 15 can be described by a history $H = [o_1, o_2]$ where $o_1 = \text{total}_1(\text{Alice}):1900$, $o_2 = \text{transfer}_2(a, b, 100)$. History $I = [o_1, o_2, o_3]$ is a continuation of H where operation $o_3 = \text{total}_3(\text{Alice}):1800$. Subhistories of I are: $[], [o_1], [o_2], [o_3], [o_1, o_2], [o_1, o_3], [o_2, o_3], [o_1, o_2, o_3]$. Prefixes of I are $[], [o_1], [o_1, o_2]$, and $[o_1, o_2, o_3]$. $\square$

Looking closely at the subhistories presented in Example 16, we see that not all operations in all subhistories are legal. For instance, in subhistory $[o_1, o_3]$, the operation $\text{total}_3(\text{Alice}):1800$ is not legal because the second *total* operation o_3 does not return the correct value if the *transfer* operation o_2 is removed from the history. Thus, the subhistory $[o_1, o_3]$ can not be considered as a “correct” subhistory of H .

To distinguish subhistories that contain only legal operations from those that contain illegal operations, we extend the notion of legal operations to *legal histories* by applying Definition 12 to finite operation sequences. Legality of histories is crucial for the definition of different commutativity relations in Section 3.3 and is the correctness criteria of the history merging mechanism described in Section 3.5.

DEFINITION 17 (LEGAL HISTORY)

A history $H = [o_1, \dots, o_n]$ is called *legal for a state S* iff there exists a sequence of states $S_0, \dots, S_{n-1}$, with $S = S_0$, where all operations o_i are legal in S_{i-1} , for $i = 1, \dots, n$, and $S_i = \Pi_1(\text{type}(o_i)(S_{i-1}, \text{input}(o_i)))$, for $i = 1, \dots, n$, i.e., S_i results from the execution of operation o_i in state S_{i-1} . We say a history H is *legal* iff H is legal for state $S_\perp$, i.e., it is legal for the empty state.

Note that the latter formulation can be used as a shorthand for the former, since history $I = J \bullet H$ is legal iff history H is legal in state S with $S = J(S_\perp)$.

Example 17 (Legal and illegal histories) Consider the subhistories of $I = [o_1, o_2, o_3]$ as described in Example 15. The following subhistories of I are legal in state S where $S(a) = (\text{Alice}, 1000)$, $S(b) = (\text{Bob}, 900)$, and $S(c) = (\text{Alice}, 900)$: $[], [o_1], [o_2], [o_1, o_2], [o_2, o_3]$, and $[o_1, o_2, o_3]$. Note that the empty history $[]$ is legal for all possible states $S \in \mathbb{S}$. The histories $[]$ and $[o_1]$ do not alter the database state, i.e., $[](S) = S$ and $[o_1](S) = S$. All the other legal subhistories alter the database state to T , i.e., $[o_2](S) = [o_1, o_2](S) = [o_2, o_3](S) = [o_1, o_2, o_3](S) = T$ with $T(a) = (\text{Alice}, 900)$, $T(b) = (\text{Bob}, 1000)$, and $T(c) = (\text{Alice}, 900)$. The subhistories $[o_3]$ and $[o_1, o_3]$ of I are *not* legal in state S , since o_3 returns 1900 instead of 1800. The history $[o_3]$ is legal for state T , whereas $[o_1, o_3]$ is illegal for any state $U \in \mathbb{S}$. $\square$

In contrast to our definition, Herlihy and Weil define legal histories independently of a state using *serial specifications* of objects [Her86, Wei88a, HW88, Wei89b, Her90, Wei93, LMWF94]. There, serial specifications are automata defined over operations. A history is considered legal only if it is accepted by the serial specifications of the objects in the database. This approach has several drawbacks. First, it is not realistic to specify such an automaton for each object (type) in the database. Second, this approach can only be used if the effect of

an operation type is local to a single object (cf. the discussion in Section 3.2.2). Third, their concept of serial specifications implicitly encodes an initial database state, which is fixed. This precludes to express that a history is legal for a *specific* database state. An advantage of serial specifications is that arbitrary constraints can be imposed on legal histories since automata are used to define legality.

In COACT, we define legality dependent on a workspace state. A history is legal if all operations in the history behave according to their specifications, i.e., their operation types, with respect to the result values externalized to a user or application program. An implication is that *every interactively executed, i.e., generated, sequence of operations is legal* (if the operation types satisfy Definition 6 and are implemented correctly). Thus, we do not have to implement a mechanism that checks legality of histories at runtime. We only have to check legality in case exchange meta-operations are invoked in COACT, i.e., when histories are merged and, hence, operations are re-executed.

The next lemma states some (trivial) facts about the relationships of legal and illegal histories and their prefixes and continuations.

LEMMA 1 Let H be a legal history (for state S). Then, any prefix I of H is also a legal history (for state S). Let H be a history that is *not* legal (for state S). Then, any continuation I of H is also an illegal history (for state S).

Proof (Lemma 1): The lemma follows immediately from Definitions 16 and 17. $\square$

The notion of legal histories allows us to define *consistency* of a workspace state and the *equivalence* of states. We consider a state as consistent if it can be produced by a legal history. Two states are called equivalent if they are not distinguishable by any further history. Formally, this equivalence is defined by requiring that the operations in a history starting from the states exhibit the same legal behavior, i.e., output parameter values.

DEFINITION 18 (CONSISTENT STATE)

A state $S \in \mathbb{S}$ is *consistent* iff there exists a legal history H with $S = H(S_{\perp})$.

DEFINITION 19 (EQUIVALENCE OF STATES)

Two states $S_1, S_2 \in \mathbb{S}$ are *equivalent* ($S_1 \equiv S_2$) iff for all histories H the following holds: H is legal in state $S_1 \Leftrightarrow H$ is legal in state S_2 .

The notion of state equivalence is more general than the notion of equality of states (cf. Lemma 2). This generalization is important: it is reasonable for a system to collect information for every operation executed (e.g., log entries for operations). As long as such information cannot affect the behavior of subsequent operations, this information should not be considered as an essential part of the database state. Definition 19 is chosen such that a state is equivalent to any other state that differs only with respect to inessential information. Thus, system internal details like the concrete state of a B-Tree index can be neglected in our model as long as no difference can be determined by an operation available to the user.

LEMMA 2 States that are equal are also equivalent: $\forall S_1, S_2 \in \mathbb{S} : S_1 = S_2 \Rightarrow S_1 \equiv S_2$.

Proof (Lemma 2): The lemma follows immediately from Definition 19 using Definition 5. $\square$

Next, we extend the definition of state equivalence to equivalence of histories. Two histories are *equivalent* for a state S if the resulting states produced by the histories are equivalent. Note that it is not required in the definition that the histories are legal.

DEFINITION 20 (EQUIVALENCE OF HISTORIES)

Two histories H_1 and H_2 are *equivalent for a state* $S \in \mathbb{S}$ ($H_1 \equiv_S H_2$) iff $H_1(S) \equiv H_2(S)$. Two histories H_1 and H_2 are *equivalent* ($H_1 \equiv H_2$) iff $H_1(S_\perp) \equiv H_2(S_\perp)$.

Note, both formulations in Definition 20 can be substituted by the other, since $I \bullet J_1 = H_1 \equiv H_2 = I \bullet J_2$ iff $J_1 \equiv_S J_2$, where $S = I(S_\perp)$.

LEMMA 3 Equivalence of states ($\equiv$) and equivalence of histories ($\equiv_S$ and $\equiv$) are *equivalence relations*, i.e., they are reflexive, symmetric, and transitive.

Proof (Lemma 3): The lemma follows immediately from Definitions 19 and 20. $\square$

Example 18 (Equivalence of states and histories) Assume that only the following operation types are available for the ACCOUNTS database (cf. Example 12): *createAccount*, *credit*, *debit*, *transfer*, *owner*, *total*, and *deleteAccount*; however, we assume there is *no balance* operation type for account data objects available.

Let S be $S(a) = (\text{Alice}, 1000)$, $S(b) = (\text{Bob}, 900)$, and $S(c) = (\text{Alice}, 900)$. Consider the following history $H = [o_1, o_2, o_3]$ where $o_1 = \text{total}_1(\text{Alice}):1900$, $o_2 = \text{transfer}_2(a, c, 100)$, and $o_3 = \text{total}_3(\text{Alice}):1900$. The resulting state is $T = H(S)$ with $T(a) = (\text{Alice}, 900)$, $T(b) = (\text{Bob}, 900)$, and $T(c) = (\text{Alice}, 1000)$. It is easy to see that state S is not equal to state T , i.e., $S \neq T$. Nevertheless, state S is equivalent to state T , i.e., $S \equiv T$, since there is no operation type available that can retrieve the balance of a single account data object. Thus, there exists no history that can distinguish state S from state T . Since no *balance* operation type is available, all subhistories of H are equivalent.

If the *balance* operation type is available for ACCOUNTS, then $S \equiv T$ is no longer true. The reason is that with a *balance* operation invocation, we can directly obtain the balance of a single account data object. Thus, there exists a history I , e.g., $I = [\text{balance}(a):1000]$, which is legal for state S but not legal for state T . The following equivalences hold in this case for the subhistories of H : $[o_1, o_2, o_3] \equiv [o_1, o_2] \equiv [o_2, o_3] \equiv [o_2]$ and $[o_1, o_3] \equiv [o_1] \equiv [o_3] \equiv []$. $\square$

This example clearly shows that Definitions 18, 19, and 20 (and most of the definitions, lemmas, propositions, and theorems in the remainder of this chapter) are dependent on the available operation types in the system. For example, all the commutativity relationships are based on the notion of equivalence of histories and, hence, equivalence of states, instead of equality of states. One may consider the dependency on the operation types in a system as a disadvantage, but our approach allows us to abstract from all irrelevant details of a state and

focus only on the relevant information that is externalized by operations to the outside world. However, to make the following definitions, etc. independent of the available operation types, it would be possible to replace the equivalence of states, i.e., $H_1(S) \equiv H_2(S)$, in Definition 20 by equality of states, i.e., $H_1(S) = H_2(S)$. A more elegant solution is to introduce an imaginary additional operation type *readState()* that returns the full state of all objects existing in the workspace.

We conclude this section by describing an important observation that is used in the remainder of this chapter to prove properties of the commutativity relations and the correctness of history merging. The proposition shows that any prefix of a legal history can be replaced by an equivalent legal prefix.

PROPOSITION 1

Let $H = I \bullet J$ be a legal history (in state T), K be a legal history (in state T), and $I \equiv K$ ($I \equiv_T K$). Then, $K \bullet J$ is legal (in state T) and $K \bullet J \equiv H$ ($K \bullet J \equiv_T H$).

Proof (Proposition 1): The proof makes use of Definitions 19 and 20. From $I \equiv K$ ($I \equiv_T K$) it follows that $I(T) \equiv K(T)$ (T might be $S_\perp$).

History J is legal (in state $I(T)$), since H is legal (in state T). From $I(T) \equiv K(T)$ it follows that J is legal in state $K(T)$. Hence, $K \bullet J$ is legal (in state T).

Because $I(T) \equiv K(T)$, no legal history $J \bullet L$ can detect a difference between $I(T)$ and $K(T)$. Thus, no legal history L can detect a difference between $(I \bullet J)(T)$ and $(K \bullet J)(T)$. From this it follows that $K \bullet J \equiv H$. $\square$

3.3 Commutativity

Many concurrency control techniques are based on a simple intuition. If the order in which two operations are executed does not affect the result, then a transaction processing system should be able to allow different transactions to perform those operations concurrently. Informally, two operations *commute* (do not conflict) if their effects on the state of the data object and their return values are the same regardless of their execution order. This ensures that no transaction can observe a difference between both execution orders.

Commutativity relations have been extensively studied in the area of concurrency control and recovery for DBMSs. Initial work in that area left the data uninterpreted, or viewed transactions as sequences consisting of simple reads and writes to disk pages. In the last 15 years, a number of researchers have considered to exploit more of the *semantics of the data* accessed by transactions or the *semantics of operations* a transaction is composed of [Kor83, SS84, BBG89, Wei91, O'N86]. For example, Weihl et al. have shown how the specifications of abstract data types can be used to permit higher levels of concurrency [WL85, Wei88a], by designing type-specific concurrency control algorithms that take advantage of algebraic properties of a type's operations. Such techniques are also useful in distributed systems and in object-oriented database systems [MRW⁺93] (see also Sections 2.3.1 and 2.3.2).

Almost all types of current transaction processing systems employ to some extent semantic knowledge about operations and objects in the database. According to [RC97], the concurrency and recovery semantics of data objects depends on the following.

- *Operation semantics* is the most commonly used semantics in the context of transaction management. With operation semantics we (1) refer to the effects of an operation to the state of objects in a database and (2) the interpretation of the input parameter values and the output parameter values of an operation.
- *Object organization semantics* refers to the abstract organization of complex data objects. This category can be further classified as *composition* semantics, which pertains to what an object is composed of, and as *order* semantics, which refers to the relative ordering among the component objects.

Most concurrency control and recovery algorithms for search structures like B-Trees [BS77, SG88, Moh96] exploit object organization semantics. Other examples are multi-granular locking [GLPT75], multi-level transactions [Wei86, Wei88b, Wei91], and fragmentable and reorderable objects [WC95].

- *Usage semantics* refers to how data objects are used and what is done with the information extracted from data objects by a transaction. The usage semantics is dependent on the structural and behavioral semantics of the transactions in the system.

Advanced transaction models that exploit the usage semantics of data objects are, for example, CAD transactions [BKK85, KKB87], cooperative transaction groups [Ska89, NZ90, NSZ91, NRZ92], Split/Join transactions [PKH88, KP92], SAGAs [GMS87], Flex transactions [BEK93], the DOM transaction model [BOH⁺92], transactions with breakpoints [FÖ89], and the nested transaction model used in the Open Publication Environment [MRKN92].

The first category, namely *operation semantics*, has primarily been used to define various forms of *commutativity of operations*, which determine semantically whether two operations are allowed to execute concurrently without compromising serializability, the traditional database correctness criterion [Pap86, BHG87].

In the COACT model, we use commutativity relations among operations to describe the semantics of operations within a cooperative activity. In contrast to semantics-based concurrency control, our focus is not in enhancing concurrency among concurrently executing transactions. Our goal is to enable the semantically correct information exchange among concurrently working users without compromising the legality of the workspace histories. We exploit the commutativity relations in two ways in COACT.

1. *Intra-workspace dependencies*: Commutativity is used to identify dependencies between operations executed in the *same* workspace. Using these dependencies, we are able to dynamically compute *consistent units of work* that can be exchanged among workspaces

and to determine the minimal sequence of operations to be compensated in case a merge conflict occurs. Additionally, this allows us to implement a selective undo mechanism for private workspaces.

2. *Inter-workspace conflicts*: Commutativity is used to identify conflicts among operations executed in *different* workspaces. This is needed when an exchange meta-operation is invoked, i.e., when workspace histories are merged. To ensure the correctness of the merge, i.e., the legality of the resulting history, a choice has to be taken among conflicting operations from different workspaces.

Weihl has shown [Wei88a, HW88, Wei89a, Wei93, LMWF94] that the type of commutativity relation applicable for concurrency control is dependent on the recovery policy used and/or the strategy that is applied in propagating updates to the stable database (cf. Section 1.1).

One update policy is called *update-in-place* [HR83]. Update-in-place are all models where a single “current” database state is maintained. When an operation is executed, the current state is used to determine the output parameter values of the operation and is modified to reflect any changes to the state performed by the operation. When a transaction commits, nothing needs to be done, since the current database state already reflects the effects of the transaction's operations. When a transaction aborts, however, the effects of the transaction's operations must be undone in some fashion. For this, *log-based recovery algorithms* are applied.

The second update policy is called *deferred-update* [HR83]. Deferred-update are all models in which the permanent database state is not updated until a transaction commits and the transaction's operations are recorded in so-called *intention lists*. Alternatively, one can think of each transaction as having its own *private workspace* (!) with the copy of the database in which it makes changes; these changes are not seen by other transactions until it commits. Transaction aborts are simple to handle in deferred-update models, since intention lists or private workspaces can simply be discarded.

In case of update-in-place policies, *backward commutativity* [Wei88a, LMWF94] has to be applied for concurrency control algorithms. If deferred-update is used as propagation strategy, we have to apply *forward commutativity* [Wei88a, LMWF94] to implement concurrency control protocols. Forward commutativity is also applicable to replicated data objects [Her90]. Both types of commutativity relations differ in a subtle but substantial way. Backward and forward commutativity are incomparable in the sense that there exist operation pairs that commute forward but not backward and other operation pairs that commute backward but not forward [Wei88a, LMWF94].

In the COACT model, we apply an update-in-place policy within workspaces. This means that operations that are executed by a user or re-executed by the system directly affect the state of the data objects in a workspace. Thus, for determining intra-workspace dependencies, we have to consider backward commutativity. If an information exchange is initiated in COACT, different workspace histories are merged, which results in a re-execution of (some) operations in a different workspace. This situation corresponds to a deferred-update policy. Hence, for

determining conflicts between operations generated in different private workspaces, we have to consider forward commutativity.

It is important to note that both commutativity relations are defined as binary relations on operations, and not on operation invocations. This means that we can use the input *and* output parameter values of operations to specify the commutativity relations. As opposed to pessimistic concurrency control algorithms, in COACT it is easily possible to exploit the return values of operations in the specification of the commutativity relations. The reason is that no blocking concurrency control mechanism is applied among the operations in the same or different workspaces, since the objects in the workspaces are replicated. Only when an exchange meta-operation is invoked the commutativity relations are used to guarantee the semantical correctness of an information exchange. At this point, the return values of all operations in the system are known.

In the following, we formally define the different commutativity relations and present their basic properties. These properties are later used to describe the correctness of our history merging approach and the merge algorithms.

3.3.1 Commutativity Relations for Intra-Workspace Dependencies

First we look at commutativity relations that are used for computing intra-workspace dependencies among operations. Since this corresponds to an update-in-place policy, we use the backward commutativity relation bc , which is a symmetric relation by definition.

After introducing the bc relation, we discuss its asymmetric variant called *right backward commutativity* (rbc) [Wei89a, AA94]. The rbc relation is used in the generalization of the basic COACT model to asymmetric relations (cf. Section 3.5.5).

3.3.1.1 Backward Commutativity

Two operations commute backward if whenever one ordering of the operations is legal, the other ordering of the operations is legal, too, and both resulting states are equivalent. This means that the output values of operations are not changed by reordering the operations in a workspace history nor can subsequent operations in the workspace detect a difference between the two operation orders. The motivation for the name “backward commutativity” is that whenever two operations can be executed directly after each other, the second operation can be pushed “backward” over the first operation [Wei93].

DEFINITION 21 (BACKWARD COMMUTATIVITY RELATION)

The *backward commutativity* relation $bc \subseteq \Omega \times \Omega$ is defined as follows: $\forall o, p \in \Omega : (o, p) \in bc$ iff $\forall$ histories $H : (H \bullet o \bullet p \text{ legal} \vee H \bullet p \bullet o \text{ legal}) \Rightarrow (H \bullet o \bullet p \text{ legal} \wedge H \bullet p \bullet o \text{ legal} \wedge H \bullet o \bullet p \equiv H \bullet p \bullet o)$.

The definition of backward commutativity is illustrated in Figure 3.2. The solid lines indicate an operation order to be legal and the dashed lines indicate another legal operation se-

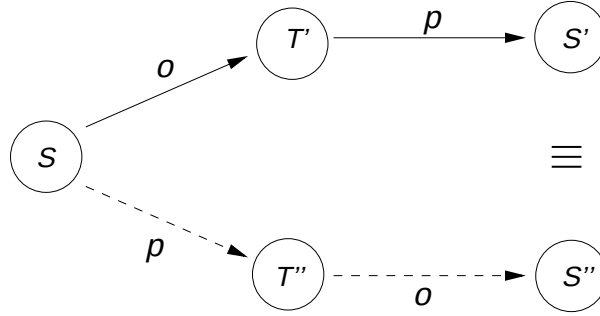


Figure 3.2: Backward Commutativity.

quence. If one of these sequences is legal, then the other has to be legal, too, and the resulting states S' and S'' have to be equivalent.

We say two operations o and p *conflict backwards* if o and p do not commute backward. With $\overline{bc}$, we denote the complement of the bc relation, i.e., the backward conflict relation. The relation bc is symmetric but *not* transitive. An example for the non-transitivity of bc is given in Example 20.

LEMMA 4 The relation bc is symmetric, i.e., $\forall o, p \in \Omega : (o, p) \in bc \Leftrightarrow (p, o) \in bc$.

Proof (Lemma 4): The lemma follows directly from Definition 21. □

Commutativity relations are usually specified by commutativity tables [BBG⁺83, Kor83, SS84]. The specification of the commutativity relation provides for a pair of partially instantiated operation types a predicate involving their parameters or parameter values. By applying the specified predicates to the respective operation instances, the membership in the commutativity relation can be easily checked at runtime. Note that the evaluation of the predicates depends only on those parts of the workspace state reflected in the parameters.

Example 19 (Backward commutativity relation for ACCOUNTS) Table 3.1 shows the backward commutativity table for the operations available in ACCOUNTS (c.f. Example 12). Some entries in the commutativity table require further explanation:

- (1): We assume that no application can see the system-internal OID of the two account objects created.
- (2), (2'): The state produced by $[o, p]$ is not equivalent to the state produced by $[p, o]$, since an account object cannot be credited, etc., before it is created.
- (3), (3'): The state produced by $[o, p]$ is not equivalent to the state produced by $[p, o]$, since an account data object cannot be deleted before it is created.
- (4), (4'): Because we do not know if $\Pi_1(S(d_1)) = n_2$ (or $\Pi_1(S(d_2)) = n_1$), we have to assume that the returned value of the *total* operation differs in both execution orders. The reason is that no information about the qualifying objects is returned by *total*.

Table 3.1: Backward Commutativity Relation for ACCOUNTS.

First operation (o)	Second operation (p)	$createAccount(n_2);d_2$	$credit(d_2, a_2)$	$debit(d_2, a_2)$	$transfer(d_2, e_2, a_2)$	$owner(d_2);n_2$	$balance(d_2);a_2$	$total(n_2);a_2$	$deleteAccount(d_2)$
$createAccount(n_1);d_1$		$true^1$	$d_1 \neq d_2^2$	$d_1 \neq d_2^2$	$d_1 \neq d_2^2$	$d_1 \neq d_2^2$	$d_1 \neq d_2^2$	$d_1 \neq d_2^2$	$d_1 \neq d_2^3$
$credit(d_1, a_1)$		$d_1 \neq d_2^{2'}$	$true$	$true$	$true$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$a_1 = 0^4$	$d_1 \neq d_2^5$
$debit(d_1, a_1)$		$d_1 \neq d_2^{2'}$	$true$	$true$	$true$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$a_1 = 0^4$	$d_1 \neq d_2^5$
$transfer(d_1, e_1, a_1)$		$d_1 \neq d_2^{2'}$	$true$	$true$	$true$	$true$	$(d_1 \neq d_2 \wedge e_1 \neq d_2) \vee a_1 = 0$	$a_1 = 0^4$	$d_1 \neq d_2^5$
$owner(d_1);n_1$		$d_1 \neq d_2^{2'}$	$true$	$true$	$true$	$true$	$true$	$true$	$d_1 \neq d_2^5$
$balance(d_1);a_1$		$d_1 \neq d_2^{2'}$	$d_1 \neq d_2 \vee a_2 = 0$	$d_1 \neq d_2 \vee a_2 = 0$	$(d_1 \neq d_2 \wedge d_1 \neq e_2) \vee a_2 = 0$	$true$	$true$	$true$	$d_1 \neq d_2^5$
$total(n_1);a_1$		$d_1 \neq d_2^{2'}$	$a_2 = 0^4$	$a_2 = 0^4$	$a_2 = 0^4$	$true$	$true$	$true$	$d_1 \neq d_2^5$
$deleteAccount(d_1)$		$d_1 \neq d_2^{3'}$	$d_1 \neq d_2^{5'}$	$d_1 \neq d_2^{5'}$	$d_1 \neq d_2^{5'}$	$d_1 \neq d_2^{5'}$	$d_1 \neq d_2^{5'}$	$d_1 \neq d_2^{5'}$	$d_1 \neq d_2$

(5), (5'): The state produced by $[o, p]$ is not equivalent to the state produced by $[p, o]$, since an account data object cannot be credited, etc. after it is deleted. $\square$

Note, almost all the commutativity relationships found in the literature do not consider the creation and deletion of data objects—they just assume that the objects already exist since they allow only operations defined on *single* objects (cf. Section 3.2.1). An exception is the work described in [AA94] which considers not only operations on single instance objects but also operations on classes, i.e., extensions, in the database.

In our model, the operation types are not restricted to access or modify only single data objects nor are they assigned to specific object types or classes. Therefore, operation types that create and delete data objects, e.g., *createAccount* and *deleteAccount*, are considered as ordinary operations in the commutativity relations. This allows a unified treatment of all operation types defined in a system. The next example gives a more detailed insight into the nature of backward commutativity.

Example 20 (Backward commutativity relation for ACCOUNTS-2 database) We introduce a new example database ACCOUNTS-2 by redefining the operation types *debit* and *transfer* of Example 12 (page 50) as follows:

***debit*(OBJECT *account*, REAL⁺ *amount*):[BOOL *success*]** decrements the balance of *account* by *amount* only if the resulting balance is greater or equal 0:

$$\begin{aligned} & \text{debit} : \mathbb{S} \times \text{OBJECT} \times \text{REAL}^+ \rightarrow \mathbb{S} \times \text{BOOL} \text{ with} \\ & \text{debit}(S, \text{account}, \text{amount}) = \begin{cases} (S, \text{false}) & \text{if } \text{balance} \Leftrightarrow \text{amount} < 0 \\ (T, \text{true}) & \text{otherwise} \end{cases} \end{aligned}$$

where $T = S[\text{account} \mapsto (\text{name}, \text{balance} \Leftrightarrow \text{amount})]$. *Debit* returns *true* if it succeeded and *false* otherwise.

***transfer*(OBJECT *account*₁, OBJECT *account*₂, REAL⁺ *amount*)** transfers a specified *amount* of money from *account*₁ to *account*₂ if the balance of *account*₁ does not become negative.

$$\begin{aligned} & \text{transfer} : \mathbb{S} \times \text{OBJECT} \times \text{OBJECT} \times \text{REAL}^+ \rightarrow \mathbb{S} \times \text{BOOL} \text{ with} \\ & \text{transfer}(S, \text{account}_1, \text{account}_2, \text{amount}) = \begin{cases} (S, \text{false}) & \text{if } \text{balance}_1 \Leftrightarrow \text{amount} < 0 \\ (T, \text{true}) & \text{otherwise} \end{cases} \end{aligned}$$

where $T = S[\text{account}_1 \mapsto (\text{name}_1, \text{balance}_1 \Leftrightarrow \text{amount})][\text{account}_2 \mapsto (\text{name}_2, \text{balance}_2 + \text{amount})]$. *Transfer* returns *true* if the operation succeeded and *false* otherwise.

Table 3.2 depicts the backward commutativity relation *bc* for ACCOUNTS-2. The rows and columns for *createAccount* and *deleteAccount* are omitted since they are the same as in Table 3.1. This holds also for the other commutativity relations that are presented in the following.

Table 3.2 also shows that the *bc* relation is not transitive. Consider the following legal history $H = [\text{credit}(a, 10), \text{balance}(b):0, \text{debit}(a, 100):\text{true}]$. The *credit* and the *debit* operations commute backward with the *balance* operation as shown in Table 3.2, but *credit*(*a*, 10) and *debit*(*a*, 100);*true* do not commute backward. $\square$

The following proposition extends the definition of backward commutativity to finite operation sequences. It shows that the exchange of backward commuting operations in a legal history yields again a legal history that is equivalent to the original history.

PROPOSITION 2

Let H be a legal history and I be a reordering of H with $\forall o, p \in H, (o, p) \notin bc : o <_I p \Leftrightarrow o <_H p$. Then, history I is legal and $I \equiv H$.

Proof (Proposition 2): The history I can be obtained from history H by a finite sequence of reordering steps, in each of which two consecutive backward commuting operations are interchanged. The proof proceeds by induction on the number of reordering steps. If there are no such steps, then $H = I$ and the result is immediate.

Suppose the result is true for any number of reordering steps up to $k \geq 1$, and suppose that I is obtained from H by $k + 1$ reordering steps. Then there exists another history J such that J is obtained from H in k reordering steps and I is obtained from J in one reordering step. The induction hypothesis implies that J is legal and that $J \equiv H$.

Then, there exist operations o, p and histories K_1 and K_2 such that $(o, p) \in bc$, $K_1 \bullet o \bullet p \bullet K_2 = J$, and $K_1 \bullet p \bullet o \bullet K_2 = I$. The history $K_1 \bullet o \bullet p$ is legal by induction hypothesis.

Table 3.2: Backward Commutativity Relation for ACCOUNTS-2.

First operation (o)	Second operation (p)	$credit(d_2, a_2)$	$debit(d_2, a_2):true$	$debit(d_2, a_2):false$	$transfer(d_2, e_2, a_2):true$	$transfer(d_2, e_2, a_2):false$	$owner(d_2):n_2$	$balance(d_2):a_2$	$total(n_2):a_2$
$credit(d_1, a_1)$		$true$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_1 = 0$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_1 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1):true$		$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$d_1 \neq e_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_1 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1):false$		$d_1 \neq d_2$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_2 = 0$	$true$	$(d_1 \neq d_2) \wedge d_1 \neq e_2$ $\vee a_2 = 0$	$true$	$true$	$true$	$true$
$transfer(d_1, e_1, a_1):true$		$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$e_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$(d_1 \neq d_2) \wedge e_1 \neq d_2$ $\vee a_1 = 0$	$(e_1 \neq d_2) \wedge d_1 \neq e_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$(d_1 \neq d_2) \wedge e_1 \neq d_2$ $\vee a_1 = 0$	$true$	$(d_1 \neq d_2) \wedge e_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$transfer(d_1, e_1, a_1):false$		$d_1 \neq d_2$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_2 = 0$	$true$	$(d_1 \neq d_2) \wedge d_1 \neq e_2$ $\vee a_2 = 0$	$true$	$true$	$true$	$true$
$owner(d_1):n_1$		$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$
$balance(d_1):a_1$		$d_1 \neq d_2$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_2 = 0$	$true$	$(d_1 \neq d_2) \wedge d_1 \neq e_2$ $\vee a_2 = 0$	$true$	$true$	$true$	$true$
$total(n_1):a_1$		$a_2 = 0$	$a_2 = 0$	$true$	$a_2 = 0$	$true$	$true$	$true$	$true$

Since o and p commute backward, $K_1 \bullet p \bullet o$ is also legal and $K_1 \bullet o \bullet p \equiv K_1 \bullet p \bullet o$. Since $K_1 \bullet o \bullet p \bullet K_2$ is legal, Proposition 1 implies that $I = K_1 \bullet p \bullet o \bullet K_2$ is legal, too, and $J = K_1 \bullet o \bullet p \bullet K_2 \equiv K_1 \bullet p \bullet o \bullet K_2 = I$. This yields the proposition since $\equiv$ is an equivalence relation (Lemma 3) and $J \equiv H$. $\square$

3.3.1.2 Excursus: Serializability in Update-In-Place Models

In the following, we give an excursus how backward commutativity can be used to define the correctness for the concurrent processing of transactions in commercial DBMSs. These systems usually apply update-in-place together with log-based recovery. We discuss serializability here in order to relate our work to this traditional correctness criterion. In Section 3.5, we show that the COACT model allows non-serializable workspace histories while always ensuring that the workspaces remain consistent, i.e., their histories remain legal.

A transaction $T = (O_T, <_T)$ is simply considered as a totally ordered sequence of oper-

ations $o \in \Omega$, followed either by a commit (c) or an abort (a) event. Alternatively, we can treat a transaction as a partially ordered set of operations where only the backward conflicting operations are ordered, i.e., $\forall o, p \in O_T : (o, p) \notin bc \Rightarrow o <_T p \vee p <_T o$. Of course, commit or abort has to be the last “operation” in such a partially ordered transaction.

Let $T = \{T_1, \dots, T_n\}$ be a set of transactions. A *complete schedule* $S = (O_S, <_S)$ over T is a total order with ordering relation $<_S$, $O_S = \bigcup_{i=1}^n O_{T_i}$, and $<_S \supseteq \bigcup_{i=1}^n <_{T_i}$, i.e., the total order $<_S$ has to reflect all total orders $<_{T_i}$ ⁴. A *schedule* is simply a prefix of a complete schedule. Note that the operations in a schedule are always legal since their result is *computed* based on the actual state of the database and the input parameter values.

A transaction T_i is *committed* in a schedule S if the corresponding commit event $c_{T_i} \in O_S$. A transaction T_i is *aborted* in a schedule S if the corresponding abort event $a_{T_i} \in O_S$. A transaction T_i is *active* in a schedule S if it is neither committed nor aborted in S . The *committed projection* $C(S)$ of a schedule S is the history obtained by removing all commit and abort events and all operations from active and aborted transactions.

Let $T = \{T_1, \dots, T_n\}$ be a set of committed transactions. We call a history H with $op(H) = \bigcup_{i=1}^n O_{T_i}$ a *serial history* if, for any two transactions T_i and T_j , $i \neq j$, in H , either all operations of T_i appear before all operations of T_j or vice versa. Thus, a serial history represents an execution without interleaving of operations of different transactions.

We call a *history* H with $op(H) = \bigcup_{i=1}^n O_{T_i}$ (*conflict*)-*serializable in the order* $<_T$ if H is legal and it is equivalent to the legal serial history I with $\forall o_i \in O_{T_i}, o_j \in O_{T_j}, i \neq j : T_i <_T T_j \Rightarrow o_i <_I o_j$. We call a *history* H (*conflict*)-*serializable* if H is legal and it is equivalent to some legal serial history, i.e., serializable in an arbitrary order. A *schedule* S is called (*conflict*)-*serializable* if its committed projection $C(S)$ is (*conflict*)-serializable. A *schedule* is called *recoverable* if, whenever for operations $o_i \in O_{T_i}, o_j \in O_{T_j}, i \neq j$, with $o_i <_S o_j \wedge (o_i, o_j) \notin bc$, the commit event of T_i precedes the commit event of T_j .

To determine if a schedule is serializable, we can apply Proposition 2. We have to look for a reordering of the committed projection $C(S)$ of S that fulfills the conditions of Proposition 2 and that is equal to a serial history consisting of the committed transactions in S .

PROPOSITION 3 (SERIALIZABILITY FOR UPDATE-IN-PLACE MODELS)

A schedule S is serializable iff there exists a reordering of $C(S)$ that is a serial legal history and that fulfills the conditions in Proposition 2. If there exist more than one such serial histories then those serial histories are equivalent.

Proof (Proposition 3): The proposition follows immediately from the definition of serializability and Proposition 2. $\square$

Usually, the existence of such a reordering is proved by using a *serialization graph* for a schedule S . A serialization graph $SG(S)$ is a directed graph whose nodes are the committed

⁴In general, a schedule can also be a partial order. Since we consider histories as totally ordered, we need to extend our definitions to partial orders to capture this case. This is straightforward, but general concurrency control is not in the focus of this thesis.

transactions T_i in S and whose edges are $T_i \rightarrow T_j$ ($i \neq j$) such that one of T_i 's operations precedes and backward conflicts with one of T_j 's operation. A schedule S is then (conflict-)-serializable iff the serialization graph $SG(S)$ is acyclic. This is the *fundamental theorem of serializability theory* [BHG87].

In practice, the serializability of transactions is ensured by dedicated *concurrency control* protocols (cf. Section 1.1). The bc relation together with Propositions 2 and 3 can be used to define both semantics-based pessimistic and optimistic concurrency control mechanisms for update-in-place models. For pessimistic concurrency control the direct utilization of bc is complicated since the result of a blocked operation is not known until it returns. One approach to resolve this problem is to pessimistically acquire locks for all operations that are instances of the same operation invocation. After the result of the operation is known, the other locks not needed can be released [Wei89a]. Alternatively, we can acquire a generic lock for the operation invocation and, after the operation returned, acquire a specific lock for the operation instance and release the generic lock for the operation invocation. Another approach is to define a special backward commutativity relation that uses only the return value of the first operation in its definition [Rak97]. Note that the former approach subsumes the latter, thereby allowing even more concurrency.

In case the semantics of operations (beyond read and write operations) is exploited for concurrency control, the log-based *recovery protocols* get more complicated since before-images can no longer be used for recovery. Compensation operations have to be defined that semantically undo the effects of operations rather than just restoring the database state to the state directly before the operation to be backed-out (logical logging, cf. Section 1.1). In Section 3.5.3, we discuss this issue in more detail. A unified framework for recovery and concurrency control for semantically rich operations can be found in [SWY93, AVA⁺94, VHBS98]. For more detailed information and for an overview of the theory behind serializability and recovery, we refer to our paper in [dBKV98].

3.3.1.3 Right Backward Commutativity

In this section, we present the asymmetric variant of the backward commutativity relation called *right backward commutativity* rbc [Wei89a, AA94]. Like backward commutativity, rbc is defined (only) for update-in-place models. The rbc relation is motivated by the observation that in some cases two operations can be exchanged only if they appear in a specific order. The name results from the fact that whenever an operation p can be directly executed after (or “right of”) an operation o it can be pushed “backward” so that it appears before operation o (but not necessarily vice versa) [Wei93].

Example 21 (Asymmetry of right backward commutativity) Consider the following history $H = [debit(a, 100); true, credit(a, 10)]$. From $debit(a, 100); true$ we can infer that the balance of account a is greater or equal 100. Because $credit$ increases this balance, we can be sure that also $[credit(a, 10), debit(a, 100); true]$ is legal. On the other hand, from the his-

tory $[credit(a, 10), debit(a, 100):true]$ one can *not* deduce that the *debit* operation succeeds, if we exchange the operation order. Hence, $credit(a, 10)$ does commute right backward with $debit(a, 100):true$, but not vice versa. $\square$

DEFINITION 22 (RIGHT BACKWARD COMMUTATIVITY RELATION)

Operation $p \in \Omega$ commutes right backward with $o \in \Omega$, i.e., $(o, p) \in rbc \subseteq \Omega \times \Omega$ iff $\forall$ histories $H : H \bullet o \bullet p \text{ legal} \Rightarrow (H \bullet p \bullet o \text{ legal} \wedge H \bullet o \bullet p \equiv H \bullet p \bullet o)$.

We say operation p *conflicts right backwards* with operation o if p does not commute right backward with o . With $\overline{rbc}$ we denote the complement of the rbc relation, i.e., the right backward conflict relation. The next lemma shows the relationship among the bc relation and right backward commutativity.

LEMMA 5 The backward conflict relation $\overline{bc}$ is the symmetric closure of $\overline{rbc}$. In other words, $\forall o, p \in \Omega : (o, p) \in bc \Leftrightarrow (o, p) \in rbc \wedge (p, o) \in rbc$.

Proof (Lemma 5): The lemma follows immediately from Definitions 21 (page 62) and 22. $\square$

From the lemma it follows that $bc \subseteq rbc$. Like bc , the rbc relation is *not* transitive. In contrast to bc , right backward commutativity is an asymmetric relation as Example 21 shows.

Example 22 (Right backward commutativity relation for ACCOUNTS-2) Table 3.3 shows the right backward commutativity for the operations of the ACCOUNTS-2 database. The table entries that differ from the bc relation are marked with asterisks. $\square$

The next example shows an interesting observation about the right backward commutativity relation, resulting from the asymmetry of rbc .

Example 23 (“False” dependencies of the rbc relation) Consider the legal history $H = J \bullet o \bullet p$ where $o = debit_1(a, 110):false$ and $p = debit_2(a, 100):true$. Assume that both *debit* operations were generated, i.e., originally invoked by a user, in this order. Since $(o, p) \in rbc$, Definition 22 yields that $I = J \bullet p \bullet o$ is legal and equivalent to H . Now look at history I . Since $(p, o) \notin rbc$, we *cannot* deduce from I that H is legal and $H \equiv I$ using only Definition 22.

This “false” conflict results from the fact that we have lost some important information about the state $S = J(S_\perp)$ in history I . From history I , we can only infer from the output parameter values of o and p that $210 > balance(a) \geq 100$. From history H , we can infer that $110 > balance(a) \geq 100$ in state S . Thus, the output value of an invocation of $debit_1(a, 110)$ *after* $debit_2(a, 100)$ in state S yields the same output parameter values as in the opposite order. It follows that, also in history I , the operation $debit_2(a, 100):true$ does not influence the result of $debit_1(a, 110):false$ since these operations were generated in the opposite order. Only with this information, we can conclude that history H is legal. $\square$

Table 3.3: Right Backward Commutativity Relation for ACCOUNTS-2.

First operation (o)	Second operation (p)	$credit(d_2, a_2)$	$debit(d_2, a_2); true$	$debit(d_2, a_2); false$	$transfer(d_2, e_2, a_2); true$	$transfer(d_2, e_2, a_2); false$	$owner(d_2); n_2$	$balance(d_2); a_2$	$total(n_2); a_2$
$credit(d_1, a_1)$		$true$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$true *$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$true *$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); true$		$true *$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); false$		$d_1 \neq d_2$ $\vee a_2 = 0$	$true *$	$true$	$d_1 \neq e_2$ $\vee a_2 = 0 *$	$true$	$true$	$true$	$true$
$transfer(d_1, e_1, a_1); true$		$true *$	$e_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$d_1 \neq d_2 *$ $\vee a_1 = 0$	$e_1 \neq d_2 *$ $\vee a_1 = 0$	$d_1 \neq d_2 *$ $\vee a_1 = 0$	$true$	$(d_1 \neq d_2$ $\wedge e_1 \neq d_2)$ $\vee a_1 = 0$	$a_1 = 0$
$transfer(d_1, e_1, a_1); false$		$d_1 \neq d_2$ $\vee a_2 = 0$	$true *$	$true$	$d_1 \neq e_2 *$ $\vee a_2 = 0$	$true$	$true$	$true$	$true$
$owner(d_1); n_1$		$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$
$balance(d_1); a_1$		$d_1 \neq d_2$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_2 = 0$	$true$	$(d_1 \neq d_2$ $\wedge d_1 \neq e_2)$ $\vee a_2 = 0$	$true$	$true$	$true$	$true$
$total(n_1); a_1$		$a_2 = 0$	$a_2 = 0$	$true$	$a_2 = 0$	$true$	$true$	$true$	$true$

A general observation for update-in-place models is that an operation p in a history can only be influenced by (or be dependent on) operations o that were invoked and returned *before* operation p was generated in the workspace⁵. To capture this, we introduce the concept of a *view of an operation* in the following.

DEFINITION 23 (VIEW OF AN OPERATION)

Let $T = H(S_{\perp})$ be the state of the workspace at the time an operation o is *generated*⁶, where history H denotes the sequence of operations executed in the workspace. The *view of operation* o is defined as the history H , i.e., $view(o) = H$. Re-executed operations o' keep the view of their corresponding generated operation o , i.e., $view(o') = view(o)$ for all $o' = o$.

For instance, in Example 23, $view(p) = J \bullet o$ and $view(o) = J$. Since we apply update-in-place within a workspace, it follows that $o \in view(p)$, but $p \notin view(o)$. In general, if we consider only a single workspace with update-in-place (without the possibility of merging

⁵Since we consider operations as atomic and histories as totally ordered, the invocation and response events can be treated as a single event in our model.

⁶If the initial materialized state of a workspace is $S_{init} = I(S_{\perp})$, where I is a legal history (cf. Definition 30 of a cooperative activity on page 91), then $T = H(S_{init})$.

different workspaces), for two operations o and p in a history H always $o \in \text{view}(p)$ *xor* $p \in \text{view}(o)$ holds.

The idea is now to ignore all right backward conflicts among operations o and p in a legal history $J \bullet o \bullet p$, where operation o is not in the view of operation p . Thus, if p does not commute right backward with o , but o does commute right backward with p and the operations o and p were generated in the order $p < o$ in the workspace, we are allowed to exchange operations o and p without affecting the legality of the resulting history $J \bullet p \bullet o$. This is formalized in the following lemma.

LEMMA 6 Let J be a legal history, $H = o \bullet p$ a legal history in state $T = J(S_\perp)$, and $(o, p) \in rbc \vee ((p, o) \in rbc \wedge p \in \text{view}(o))$. Then, $I = p \bullet o$ is legal in state T and $I \equiv_T H$ and, hence, $J \bullet I \equiv J \bullet H$.

Proof (Lemma 6): Note that history $J \bullet H$ is legal, since J is legal, $T = J(S_\perp)$, and H is legal in state T . We prove the lemma by distinguishing two cases:

(1) If $(o, p) \in rbc$, Definition 22 yields that the history $I = J \bullet p \bullet o$ is legal, $J \bullet H \equiv J \bullet I$, and, hence, $I \equiv_T H$.

(2) If $(p, o) \in rbc \wedge p \in \text{view}(o)$, it follows from Definition 23 that operations o and p were originally generated in the order $p < o$ in the workspace with state T . Since in this execution $p < o$, the results of the legal operations o and p were computed on the actual workspace states defined by T and $T \bullet o$, the history $I = p \bullet o$ must be legal in state $T = J(S_\perp)$. Since history J is legal, $T = J(S_\perp)$, and history I is legal in state T , it follows that $J \bullet I$ is legal. Since $(p, o) \in rbc$ it follows from Definition 22 that $J \bullet I = J \bullet p \bullet o \equiv J \bullet o \bullet p = J \bullet H$ and, hence, $I \equiv_T H$. This proves the lemma. $\square$

The idea of using views of operations to define the potential dependency of operations in a history is similar to the concept of causal precedence in Lamport clocks [Lam78]. There, a “happens before” relation is employed to define dependent and concurrent operations in distributed systems. In [Lam78], an event e_1 is always dependent on event e_2 if e_2 “happened before” e_1 . In our approach, an operation p is considered as dependent on operation o if o is in the view of p and p does not commute (right) backward with o .

The following proposition extends the definition of right backward commutativity to finite operation sequences and combines it with Lemma 6.

PROPOSITION 4

Let H be a legal history and I be a reordering of H with $\forall o, p \in H, \neg((o, p) \in rbc \vee ((p, o) \in rbc \wedge p \in \text{view}(o))) : o <_H p \Leftrightarrow o <_I p$. Then, I is legal and $I \equiv H$.

Proof (Proposition 4): The proof is by an induction on the number of reordering steps, analogous to the proof of Proposition 2. Only the induction step differs in the sense that we use Lemma 6 instead of Definition 21. If there are no reordering steps, then $H = I$ and the result is immediate.

Suppose the result is true for any number of reordering steps up to $k \geq 1$, and suppose that I is obtained from H by $k + 1$ reordering steps. Then there exists another history J such that J is obtained from H in k reordering steps and I is obtained from J in one reordering step. The induction hypothesis implies that J is legal and that $J \equiv H$.

Then, there exist operations o, p and histories K_1 and K_2 such that $(o, p) \in rbc \vee ((p, o) \in rbc \wedge p \in \text{view}(o))$, $K_1 \bullet o \bullet p \bullet K_2 = J$, and $K_1 \bullet p \bullet o \bullet K_2 = I$. The history $K_1 \bullet o \bullet p$ is legal by induction hypothesis. Lemma 6 yields that $K_1 \bullet p \bullet o$ is also legal and $K_1 \bullet o \bullet p \equiv K_2 \bullet p \bullet o$. Since $K_1 \bullet o \bullet p \bullet K_2$ is legal, Proposition 1 implies that $I = K_1 \bullet p \bullet o \bullet K_2$ is also legal and $J = K_1 \bullet o \bullet p \bullet K_2 \equiv K_1 \bullet p \bullet o \bullet K_2 = I$. This yields the proposition since $\equiv$ is an equivalence relation (Lemma 3) and $J \equiv H$. $\square$

To the best of our knowledge, all previous work on right backward commutativity neglects the fact that in case an operation is “out of the view”, right backward conflicts can be neglected. The reason seems to be that in the concurrency control protocols for update-in-place models this observation is not essential. In these protocols, operations executed in the database are never really reordered in a history, e.g., undone and re-executed in different order. Operations are only hypothetically reordered to prove serializability of the schedules that pass these protocols.

In COACT, however, it is the case that the same operations are (re-)executed in different workspaces in different order when histories are merged. In order not to be over-restrictive, we can consider the view of operations to eliminate unnecessary dependencies among operations. We will come back to this issue, when we extend the basic COACT model to also include asymmetric commutativity relations (cf. Section 3.5.5).

Proposition 4 can be straightforwardly used to re(defined) the notion of serializability of transactions (cf. [Wei89a, Wei93, AA94]), but this is out of the scope of this work. Note that with rbc more schedules are serializable as with bc . In fact, it was shown in [Wei93] that pessimistic concurrency control protocols using conflict-serializability as their correctness criterion are correct only if the conflict relation used is equal or a superset to the right backward conflict relation $\overline{rbc}$. A drawback of the usage of rbc for the definition of concurrency control protocols are the more complex commit orderings induced by the asymmetry of rbc .

3.3.2 Commutativity for Inter-Workspace Dependencies

In this section, we discuss two commutativity relations that can be used for computing inter-workspace conflicts in the COACT model. Since this corresponds to a deferred-update policy, we use the forward commutativity relation fc , which is a symmetric relation by definition.

After introducing the fc relation, we discuss its asymmetric variant. We call this relation *left forward commutativity* (lfc). The lfc relation is used in the generalization of history merging to asymmetric relations (cf. Section 3.5.5).

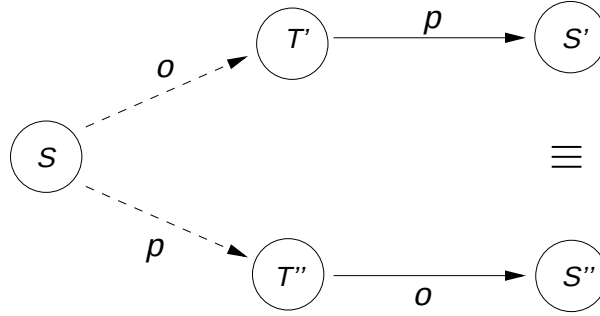


Figure 3.3: Forward Commutativity.

3.3.2.1 Forward Commutativity

Two operations commute forward if, whenever both operations are legal in a state S , both possible execution orders of the two operations are legal, too, and their resulting states are equivalent. This means that the output parameter values of operations are not changed by putting both operations into the same workspace history nor can subsequent operations in the workspace detect a difference between both operation orders. The motivation for the name “forward commutativity” is that whenever both of the two operations are executable, i.e., legal in the same state, each of them can be pushed “forward” past the other [Wei93].

DEFINITION 24 (FORWARD COMMUTATIVITY RELATION)

The *forward commutativity* relation $fc \subseteq \Omega \times \Omega$ is defined as follows: $\forall o, p \in \Omega, o \neq p : (o, p) \in fc$ iff $\forall$ histories $H : (H \bullet o \text{ legal} \wedge H \bullet p \text{ legal}) \Rightarrow (H \bullet o \bullet p \text{ legal} \wedge H \bullet p \bullet o \text{ legal} \wedge H \bullet o \bullet p \equiv H \bullet p \bullet o)$.

We say two operations o and p *conflict forward* if o and p do not commute forward. By $\overline{fc}$ we denote the forward conflict relation that is defined as the complement of fc . The definition of forward commutativity is illustrated in Figure 3.2. The dashed lines indicate operations assumed to be legal and the solid lines indicate continuations assumed to be legal. Additionally, the resulting states S' and S'' have to be equivalent⁷.

The underlying assumption in the definition of forward commutativity is that neither o nor p has already seen the effects of the other operation. In other words, both $o \notin \text{view}(p)$ and $p \notin \text{view}(o)$ hold. This is in contrast to the bc and rbc relations where either $o \in \text{view}(p)$ or $p \in \text{view}(o)$ holds, because update-in-place policy is applied in the workspace.

LEMMA 7 The relation fc is symmetric, i.e., $\forall o, p \in \Omega : (o, p) \in fc \Leftrightarrow (p, o) \in fc$.

Proof (Lemma 7): The lemma follows immediately from Definition 24. □

⁷Because histories cannot contain identical operations twice, we need the additional restriction in Definition 24 that the operations are not identical, i.e., $o \neq p$. This is not necessary for the bc and rbc relations.

Table 3.4: Forward Commutativity Relation for ACCOUNTS-2.

First operation (o)	Second operation (p)	$credit(d_2, a_2)$	$debit(d_2, a_2); true$	$debit(d_2, a_2); false$	$transfer(d_2, e_2, a_2); true$	$transfer(d_2, e_2, a_2); false$	$owner(d_2); n_2$	$balance(d_2); a_2$	$total(n_2); a_2$
$credit(d_1, a_1)$		$true$	$true^*$	$d_1 \neq d_2 \vee a_1 = 0$	$true^*$	$d_1 \neq d_2 \vee a_1 = 0$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); true$		$true^*$	$d_1 \neq d_2 \vee a_1 = 0^* \vee a_2 = 0$	$true^*$	$d_1 \neq d_2 \vee a_1 = 0^* \vee a_2 = 0$	$true^*$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); false$		$d_1 \neq d_2 \vee a_2 = 0$	$true^*$	$true$	$d_1 \neq e_2^* \vee a_2 = 0$	$true$	$true$	$true$	$true$
$transfer(d_1, e_1, a_1); true$		$true^*$	$d_1 \neq d_2 \vee a_1 = 0^* \vee a_2 = 0$	$e_1 \neq d_2^* \vee a_1 = 0$	$d_1 \neq d_2 \vee a_1 = 0^* \vee a_2 = 0$	$e_1 \neq d_2^* \vee a_1 = 0$	$true$	$(d_1 \neq d_2 \wedge e_1 \neq d_2) \vee a_1 = 0$	$a_1 = 0$
$transfer(d_1, e_1, a_1); false$		$d_1 \neq d_2 \vee a_2 = 0$	$true^*$	$true$	$d_1 \neq e_2^* \vee a_2 = 0$	$true$	$true$	$true$	$true$
$owner(d_1); n_1$		$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$
$balance(d_1); a_1$		$d_1 \neq d_2 \vee a_2 = 0$	$d_1 \neq d_2 \vee a_2 = 0$	$true$	$(d_1 \neq d_2 \wedge d_1 \neq e_2) \vee a_2 = 0$	$true$	$true$	$true$	$true$
$total(n_1); a_1$		$a_2 = 0$	$a_2 = 0$	$true$	$a_2 = 0$	$true$	$true$	$true$	$true$

Example 24 (Forward commutativity relation for ACCOUNTS-2) Table 3.4 depicts the fc relation for the ACCOUNTS-2 database. The entries that differ from the bc relation are marked with asterisks in the table.

Like bc , forward commutativity is *not* transitive. Consider the following set of operations: $debit_1(a, 10); true$, $balance_2(b); 0$, and $debit_3(a, 100); true$. Both $debit$ operations commute forward with the $balance$ operation as shown in Table 3.4, but they do not commute forward with each other. $\square$

The rather subtle differences between forward and backward commutativity can be seen by comparing Table 3.4 to Table 3.2 (page 66): the bc and the fc relations are incomparable.

LEMMA 8 There exist cases where $bc \not\subseteq fc \wedge bc \not\supseteq fc$.

Proof (Lemma 8): Example 25 shows such a case. $\square$

The reason for the incomparability of fc and bc is that we have different (implicit) information on the state $S = H(S_\perp)$ in Definitions 24 and 21 (page 62). For bc we know that whenever either $H \bullet [o, p]$ or $H \bullet [p, o]$ are legal both histories are legal, whereas for fc it is known that whenever $H \bullet o$ and $H \bullet p$ are legal both $H \bullet [o, p]$ and $H \bullet [p, o]$ are legal.

Example 25 (Incomparability of forward and backward commutativity) For instance, if history $H \bullet debit_1(a, 10):true \bullet debit_2(a, 100):true$ is legal, we can infer that in state $S = H(S_\perp)$ the balance of account a is greater or equal 110. Thus, both operations commute backward. If we know that $H_1 = H \bullet debit_1(a, 10):true$ and $H_2 = H \bullet debit_2(a, 100):true$ are legal, we can only conclude that in state $S = H(S_\perp)$ the balance of account a is greater or equal 100. Hence, we cannot guarantee that $debit_1(a, 10):true$ succeeds if it is applied in state $H_2(S_\perp)$ and, hence, both operations do not commute forward.

A similar example involves operations $o = debit_3(a, 100):true, p = debit_4(a, 110):false$. If it is known that $H \bullet o$ and $H \bullet p$ are legal, we know that in state $S = H(S_\perp)$: $110 > balance(a) \geq 100$. Therefore, o and p commute forward because $debit(110)$ will fail anyway in both possible orders. If it is known that $H \bullet [p, o]$ is legal, we can infer the same information about a . Thus, o commutes right backward with p . Operation o and p do not commute backward since from the legality of $H \bullet [o, p]$ we cannot infer that the balance of account a is less or equal 100. We can only deduce that in state $S = H(S_\perp)$ the balance of account a is greater or equal 100 which is not sufficient to conclude that $H \bullet [p, o]$ is legal. It follows that $(o, p) \notin rbc$, and, hence, $(o, p) \notin bc$. $\square$

The following proposition extends the definition of forward commutativity to a finite number of histories. It shows that the “merging” of *disjoint and legal* histories, where all operations pairwise commute forward, results again in legal histories if we preserve the order of operations in the original histories. Any merged histories constructed in this way are equivalent.

PROPOSITION 5

Let $H \bullet H_i, i = 1, \dots, n$, be legal histories with $op(O_i) \cap op(O_j) = \emptyset$, for $i \neq j$, and $\forall o \in H_i, p \in H_j, i \neq j : (o, p) \in fc$. For all histories I and J with $op(I) = op(J) = \bigcup_{i=1}^n op(H_i)$, $<_I \supseteq \bigcup_{i=1}^n <_{H_i}$, and $<_J \supseteq \bigcup_{i=1}^n <_{H_i}$, the following holds: $H \bullet I$ and $H \bullet J$ are legal and $H \bullet I \equiv H \bullet J$.

Proof (Proposition 5): We prove the proposition by induction on the number k of operations in I and J , respectively. Let $H_i = [o_{i,1}, \dots, o_{i,m_i}]$ and $k = \sum_{i=1}^n m_i$.

If $k = 1$, then all histories H_i are empty except one. Without loss of generality, assume that $H_j = [o_{j,1}]$. Then $I = J = H_j$. This proves the induction hypothesis for $k = 1$ since $H \bullet H_j$ is legal by assumption. Let $k = 2$. Then, either (1) both operations come from the same history H_i and the others are empty or (2) the operations come from different histories. In the former case, the induction hypothesis is immediate since $H_j = [o_{j,1}, o_{j,2}] = I = J$ and $H \bullet H_j$ is legal by assumption. In the latter case, without loss of generality, assume that $H_i = [o_{i,1}]$ and $H_j = [o_{j,1}], i \neq j$, and $o_{i,1} \neq o_{j,1}$. Let $I = [o_{i,1}, o_{j,1}]$ and $J = [o_{i,1}, o_{j,1}]$. These are the only two possible result histories that contain both operations. Because $o_{j,1}$ and $o_{i,1}$ commute forward, it follows that I and J are legal and $I \equiv J$. This proves the induction hypothesis.

Suppose the result holds for $k \Leftrightarrow 1 \geq 1$. We now show that the result holds for k , too. Without loss of generality, assume that $H_i = H'_i \bullet p$ and $H_j = H'_j \bullet q, i \neq j$, and $H_l = H'_l$ for $l = 1, \dots, n, l \neq i$ and $l \neq j$, for operations p and q with $p \neq q$.

By the induction hypothesis ($k \Leftrightarrow 1$ operations), it follows that (1) all histories I_1 and J_1 with $op(I_1) = op(J_1) = \{p\} \cup \bigcup_{l=1}^{l=n} op(H'_l)$ that preserve the orderings in H_i and H'_l for $l = 1, \dots, n$ are legal and pairwise equivalent and that (2) all histories I_2 and J_2 with $op(I_2) = op(J_2) = \{q\} \cup \bigcup_{l=1}^{l=n} op(H'_l)$ that preserve the orderings in H_j and H'_l for $l = 1, \dots, n$ are legal and pairwise equivalent.

Let $J_1 = M \bullet p$ and $J_2 = M \bullet q$. Because p and q commute forward by assumption, we get that $M \bullet p \bullet q = J_1 \bullet q$ and $M \bullet q \bullet p = J_2 \bullet p$ are legal and $J_1 \bullet q \equiv J_2 \bullet p$. By Proposition 1, it follows that the histories $K = I_1 \bullet q$ and $L = I_2 \bullet p$ are legal and $K \equiv L$, because $J_1 \equiv I_1$ and $J_2 \equiv I_2$.

This proves that all histories K and L (with k operations) that (1) consist of the operations in $\{q, p\} \cup \bigcup_{i=1}^{i=n} op(H'_i) = \bigcup_{i=1}^{i=n} op(H_i)$, (2) that preserve the order of operations in the histories H_l for $l = 1, \dots, n$, and (3) that end either with p or q (where $H_i = H'_i \bullet p$, $H_j = H'_j \bullet q$) are legal and pairwise equivalent. Since the histories H_i and H_j were chosen arbitrarily from the set $\{H_1, \dots, H_n\}$, this result holds for all H_i and H_j , $i \neq j$. This proves the proposition. $\square$

Forward commutativity together with Proposition 5 can be used to define serializability of transactions in deferred-update models with recovery based on intention lists and to define serializability in replicated systems [Wei88a, Wei89a, Her90]. Committed transactions simply correspond to histories H_i in Proposition 5. Because serializability and concurrency control for traditional transaction systems is out of the scope of this thesis, we do not elaborate on this issue in depth. We only give a short summary on how concurrency control and recovery can be realized in such environments.

Naturally, optimistic concurrency control is well-suited for deferred-update models with intention list-based recovery. Depending on the manner in which transactions are validated, optimistic protocols can be categorized as forward validation or backward validation [Här84]. In forward validation it is ensured that a committing transaction T cannot invalidate any active transactions $\bar{T}$ by checking for forward conflicts among operations in T and $\bar{T}$. If such transactions $\bar{T}$ do exist, either T or all transactions $\bar{T}$ have to be aborted. In backward validation, it is ensured that the committing transaction T has not been invalidated by the recent commit of other transactions $\bar{T}$. This is done by checking for forward conflicts among operations in $\bar{T}$ and T . If there exists such a conflict, transaction T cannot commit and is aborted. Semantics-based concurrency control for deferred-update models and replicated systems is discussed, e.g., in [Her86, Her90, LMWF94]. Pessimistic concurrency control schemes for deferred-update models using the fc relation can be found in [Wei88a, Wei93, LMWF94]. Here, problems occur since the computation of return values of operations in case of blocking transactions is difficult. Compared to update-in-place models, transaction recovery is easy in deferred-update models since the transactions maintain a temporary private database state, e.g., using intention lists. The operations contained in the intention lists are first applied, i.e., re-executed, in the permanent (public) database when the transaction commits. If a transaction aborts, its intention list is simply discarded.

3.3.2.2 Left Forward Commutativity

In this section, we define an asymmetric variant of forward commutativity. We call this commutativity relation *left forward commutativity* lfc . Like forward commutativity, the lfc relation is applicable (only) to deferred-update models.

Informally, operation o commutes left forward with operation p if, whenever both operations are legal in a state S , the sequence $[o, p]$ is legal, too, and both possible operation orders $[o, p]$ and $[p, o]$ result in an equivalent state. Note, that it is *not* required that the operation order $[p, o]$ is legal, too. The motivation for the name “left forward commutativity” is that whenever two operations are legal in the same state, one can be pushed “forward” before (“left”) the other in the combined history.

Example 26 (Left forward commutativity) Consider the following two operations that are legal in state $S = H(S_\perp)$ where the balance of account a is 110: $o = \text{balance}(a):110$ and $p = \text{debit}(a, 100):true$. The history $H \bullet o \bullet p$ is legal and yields 10 for the balance of account a . The history $H \bullet p \bullet o$ results also in a balance of 10 for account a . Thus, $H \bullet o \bullet p \equiv H \bullet p \bullet o$. However, $H \bullet p \bullet o$ is not legal because the operation invocation $\text{balance}(a)$ would return 10 instead of 110. $\square$

DEFINITION 25 (LEFT FORWARD COMMUTATIVITY RELATION)

Operation $o \in \Omega$ commutes left forward with operation $p \in \Omega$, i.e., $(o, p) \in lfc \subseteq \Omega \times \Omega$ iff $\forall$ histories $H : (H \bullet o \text{ legal} \wedge H \bullet p \text{ legal}) \Rightarrow (H \bullet o \bullet p \text{ legal} \wedge H \bullet o \bullet p \equiv H \bullet p \bullet o)$.⁸

We say two operations o and p *conflict left forward* if o and p do not commute left forward. By $\overline{lfc}$ we denote the left forward conflict relation, i.e., the complement of lfc . The next lemma explains the relationship between the forward commutativity relation fc and the lfc relation.

LEMMA 9 The forward conflict relation $\overline{fc}$ is the symmetric closure of $\overline{lfc}$. In other words, $\forall o, p \in \Omega : (o, p) \in fc \Leftrightarrow (o, p) \in lfc \wedge (p, o) \in lfc$.

Proof (Lemma 9): The proof follows immediately from Definitions 24 (page 73) and 25. $\square$

It follows that the forward commutativity relation fc is a subset of the lfc relation. Like forward commutativity, the lfc relation is not transitive. Left forward commutativity is an asymmetric relation as Example 26 indicates.

Example 27 (Left forward commutativity relation for ACCOUNTS-2) The left forward commutativity relation for ACCOUNTS-2 is depicted in Table 3.5. The entries in the commutativity table that differ from the fc relation are marked with a “+”. $\square$

Similar to Proposition 5 for forward commutativity, we can extend the lfc relation to an arbitrary number of disjoint and legal histories. The idea thereby is that it is allowed to also combine forward conflicting operations o and p from different histories into a single “merged”

⁸Recall that equivalence of histories does not imply legality of histories (cf. Definitions 17 and 20).

Table 3.5: Left Forward Commutativity Relation for ACCOUNTS-2.

First operation (o)	Second operation (p)	$credit(d_2, a_2)$	$debit(d_2, a_2); true$	$debit(d_2, a_2); false$	$transfer(d_2, e_2, a_2); true$	$transfer(d_2, e_2, a_2); false$	$owner(d_2); n_2$	$balance(d_2); a_2$	$total(n_2); a_2$
$credit(d_1, a_1)$		$true$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); true$		$true$	$d_1 \neq d_2 \vee a_1 = 0 \vee a_2 = 0$	$true$	$d_1 \neq d_2 \vee a_1 = 0 \vee a_2 = 0$	$true$	$true$	$d_1 \neq d_2 \vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); false$		$true^+$	$true$	$true$	$true^+$	$true$	$true$	$true$	$true$
$transfer(d_1, e_1, a_1); true$		$true$	$d_1 \neq d_2 \vee a_1 = 0 \vee a_2 = 0$	$e_1 \neq d_2 \vee a_1 = 0$	$d_1 \neq d_2 \vee a_1 = 0 \vee a_2 = 0$	$e_1 \neq d_2 \vee a_1 = 0$	$true$	$d_2 \notin \{d_1, e_1\} \vee a_1 = 0$	$a_1 = 0$
$transfer(d_1, e_1, a_1); false$		$true^+$	$true$	$true$	$true^+$	$true$	$true$	$true$	$true$
$owner(d_1); n_1$		$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$
$balance(d_1); a_1$		$true^+$	$true^+$	$true$	$true^+$	$true$	$true$	$true$	$true$
$total(n_1); a_1$		$true^+$	$true^+$	$true$	$true^+$	$true$	$true$	$true$	$true$

history I if either $(o, p) \in lfc$ or $(p, o) \in lfc$. In this case, we have to additionally consider the ordering of o and p such that both operations remain legal. For example, if $(o, p) \in lfc \wedge (p, o) \notin lfc$, we have to order o before p in I .

PROPOSITION 6

Let $H \bullet H_i, i = 1, \dots, n$, be legal histories with $op(H_i) \cap op(H_j) = \emptyset, i \neq j$, and $\forall o \in H_i, p \in H_j, i \neq j : (o, p) \in lfc \vee (p, o) \in lfc$. Let $\prec = \{(o, p) \mid \exists o \in H_i, p \in H_j, i \neq j : (o, p) \in lfc \wedge (p, o) \notin lfc\}$. Let I, J , and K be histories with $op(I) = op(J) = op(K) = \bigcup_{i=1}^n op(H_i)$, $<_I \supseteq (\prec \cup \bigcup_{i=1}^n <_{H_i})$, $<_J \supseteq (\prec \cup \bigcup_{i=1}^n <_{H_i})$, but $<_K \supseteq \bigcup_{i=1}^n <_{H_i}$. For all such histories I, J , and K it holds that $H \bullet I$ and $H \bullet J$ are legal and $H \bullet I \equiv H \bullet J \equiv H \bullet K$.

Proof (Proposition 6): The proof of the proposition is by induction on the number k of operations in $\bigcup_{i=1}^n op(O_i)$. The induction is similar to that of Proposition 5 since $(o, p) \in lfc \wedge (p, o) \in lfc \Leftrightarrow (o, p) \in fc$. Therefore, we consider here only the case where $\exists o \in O_i, p \in O_j, i \neq j : (o, p) \in lfc \wedge (p, o) \notin lfc$. Let $H_i = [o_{i,1}, \dots, o_{i,m_i}]$ and $k = \sum_{i=1}^n m_i$.

If $k = 1$, then all histories H_i are empty except one. Without loss of generality, assume that $H_j = [o_{j,1}]$. Then $I = J = K = H_j$. This proves the induction hypothesis for $k = 1$ since $H \bullet H_j$ is legal by assumption.

Let $k = 2$. Then, either (1) both operations come from the same history H_i and the other histories $H_j, i \neq j$, are empty or (2) the operations come from different histories. In the former case, the induction hypothesis is immediate since $H_j = [o_{j,1}, o_{j,2}] = I = J = K$ and $H \bullet H_j$ is

legal by assumption. In the latter case, without loss of generality, assume that $H_i = [o_{i,1}]$ and $H_j = [o_{j,1}]$, $i \neq j$, and $o_{i,1} \neq o_{j,1}$. Let $I = [o_{i,1}, o_{j,1}]$ and $J = [o_{j,1}, o_{i,1}]$. These are the only two possible merged histories that contain both operations. For the case where $(o_{i,1}, o_{j,1}) \in fc$, see Proposition 5. If $(o_{i,1}, o_{j,1}) \in lfc \wedge (o_{j,1}, o_{i,1}) \notin lfc$, we get $\prec = \{(o_{i,1}, o_{j,1})\}$. I is the only history that fulfills the conditions $<_I \subseteq (\prec \cup \bigcup_{i=1}^{i=n} <_{H_i})$ of the proposition. By the definition of left forward commutativity, $H \bullet I$ legal and $H \bullet I \equiv H \bullet J$. Since $K = J$ in this case, we get $H \bullet I \equiv H \bullet K$. This yields the induction hypothesis for $k = 2$.

The induction step is analogous to that in Proposition 5 except that we need to consider additional orderings among operations o and p if $(o, p) \in lfc \wedge (p, o) \notin lfc$. $\square$

The lfc relation together with Proposition 6 can be straightforwardly used to re(defined) the notion of serializability of transactions in deferred-update models. As an example, consider the following transactions $T_1 = [balance(a):100]$ and $T_2 = [credit(a, 10)]$ which are both based on the same permanent database state S . It is possible to serialize T_1 before T_2 without affecting the correctness of the resulting database state and the values returned by the operations since $balance(a):100$ commutes left forward with $credit(a, 10)$. However, with forward commutativity the transactions would not have been serializable since $balance(a):100$ does not commute forward with $credit(a, 10)$. Hence, with the asymmetric lfc relation more schedules are serializable compared to the symmetric fc relation but additional commit orders among transactions are induced by lfc which complicates concurrency control policies. This is similar to the relationship among bc and rbc .

To our knowledge, the asymmetric lfc commutativity relation has never been (explicitly) mentioned in the literature to define concurrency control algorithms for deferred-update models. In [Her86, HW88, Her90], serial dependency relations are used to define optimistic concurrency control protocols for deferred-update models. Since lfc is like fc a special case of serial dependency, these algorithms can be directly exploited with lfc , too.

3.3.3 Commutativity Relations Independent of the Update-Model

So far, we have defined four commutativity relations: two of them, i.e., bc and rbc , are applicable to update-in-place models and two, i.e., fc , lfc , are applicable to deferred-update models. In this section, we introduce two other types of commutativity relations that are applicable to *both* update-in-place and deferred-update policies. With such commutativity relations, the definition of concurrency control protocols is simplified since the relations are applicable to both update propagation models.

A straightforward way to define a commutativity relation that is independent from the underlying update and recovery model is to combine the bc and fc relations into a new symmetric commutativity relation c .

DEFINITION 26 (MODEL-INDEPENDENT COMMUTATIVITY RELATION)

The *model-independent commutativity* relation c is defined as $c = bc \cap fc$.

The *model-independent conflict* relation $\bar{c}$ is defined as the complement of c . Like the relations bc and fc , the commutativity relation c is a symmetric relation (but not necessarily transitive).

LEMMA 10 The model-independent commutativity relation c is symmetric, i.e., $\forall o, p \in \Omega : (o, p) \in c \Leftrightarrow (p, o) \in c$.

Proof (Lemma 10): The lemma follows immediately from the symmetry of forward commutativity (Lemma 7, page 73) and backward commutativity (Lemma 4, page 63). $\square$

Since the relation c is defined as the intersection of bc and fc , it holds that $c \subseteq bc$ and $c \subseteq fc$ which means that less operation pairs do commute. Thus, we obtain more conflicts or dependencies among operations. On the other hand, the c relation does *not* reflect the subtle differences between bc and fc . Hence, we can use the c relation independently of the update propagation model applied. Another consequence is that all the results we obtained for the bc and fc relations are also applicable to the model-independent commutativity relation c .

All of the commutativity relations introduced so far use the output parameter values of operations in their definitions. This means that they use implicit information about the workspace state contained in the return values of the operations. Thus, we can characterize these commutativity relations as *state-dependent* commutativity relations. Note that many papers published in the area of transaction management consider only commutativity relations as state-dependent that refer explicitly to specific database states and not implicitly via the return parameter values. Since it is theoretically possible to return the whole database state as a result of each operation, it is easy to see that there is no real difference between the two notions of state-dependent commutativity relations.

In the following, we present a commutativity relation called *state-independent commutativity* or *invocation-based commutativity* [Wei93]. Its definition is only based on operation invocations, which means that we *cannot* exploit any information about the outcome of the operations, i.e., the instances of the operation invocations. In other words, all operations with the same invocations have the same conflicts. Since information about the database state can only be deduced from return values of operations, the following definition is considered *state-independent*.

Informally, two operation invocations commute invocation-based if the operation invocations return the same output parameter values, regardless in which order the operations are invoked, and if the resulting states are equivalent in both execution orders. The invocation-based commutativity relation ic is then derived from the invocation-based commutativity relation for operation invocations.

DEFINITION 27 (INVOCATION-BASED COMMUTATIVITY RELATION)

The *invocation-based commutativity* relation for operation invocations $ic_{oi} \subseteq \Sigma \times \Sigma$ is defined

as follows⁹: $\forall i = j \in \Sigma : (i, j) \in ic_{oi}$ iff $\forall$ legal histories H with $S = H(S_\perp)$:

$$\begin{aligned} \Pi_1(ot_j(\vec{in}_j)(\Pi_1(ot_i(\vec{in}_i)(S)))) &\equiv \Pi_1(ot_i(\vec{in}_i)(\Pi_1(ot_j(\vec{in}_j)(S)))) \wedge \\ \Pi_{2,\dots,n+1}(ot_i(\vec{in}_i)(S)) &= \Pi_{2,\dots,n+1}(ot_i(\vec{in}_i)(\Pi_1(ot_j(\vec{in}_j)(S)))) \wedge \\ \Pi_{2,\dots,m+1}(ot_j(\vec{in}_j)(S)) &= \Pi_{2,\dots,m+1}(ot_j(\vec{in}_j)(\Pi_1(ot_i(\vec{in}_i)(S)))) \end{aligned}$$

where n and m are the numbers of output parameters of i and j .

The *invocation-based commutativity* relation for operations $ic \subseteq \Omega \times \Omega$ is defined as follows:
 $\forall o, p \in \Omega : (o, p) \in ic$ iff $\exists i, j \in \Sigma : o$ is an instance of $i \wedge p$ is an instance of $j \wedge (i, j) \in ic_{oi}$.

We say two operations o and p *conflict invocation-based* if $(o, p) \notin ic$. The *invocation-based conflict relation* $\overline{ic}$ is defined as the complement of the invocation-based commutativity relation. Like the commutativity relations fc , bc , and c , the invocation-based commutativity relation ic is symmetric.

LEMMA 11 The invocation-based commutativity relation ic is symmetric, i.e., $\forall o, p \in \Omega : (o, p) \in ic \Leftrightarrow (p, o) \in ic$.

Proof (Lemma 11): The lemma follows directly from Definition 27. $\square$

Example 28 (Invocation-based commutativity relation for ACCOUNTS-2) The invocation-based commutativity relation for ACCOUNTS_2 is depicted in Table 3.6. $\square$

The following lemma explains the relationship among invocation-based commutativity and the other kinds of commutativity relations introduced.

LEMMA 12 If two operations do commute invocation-based, then they also commute model-independent, i.e., $ic \subseteq c$.

Proof (Lemma 12): Let $o = i:\vec{r}$ and $p = j:\vec{s}$ be operations that are instances of the operation invocations i and j and that return output values $\vec{r}$ and $\vec{s}$, respectively. Assume, $(o, p) \in ic$. It follows that the operation invocations $(i, j) \in ic_{oi}$. Since the result values of operation invocations are computed based on the actual state, it follows that $H \bullet o \bullet p$ and $H \bullet p \bullet o$ are legal and $H \bullet o \bullet p \equiv H \bullet p \bullet o$. Hence, $(o, p) \in bc$ and $(o, p) \in fc$. Since c is defined as $bc \cap fc$, it follows that $(o, p) \in c$. This proves the lemma. $\square$

Since $ic \subseteq c$, all lemmas and propositions applicable to fc and bc can be applied to define concurrency control algorithms that are based on ic . Even though these algorithms permit less concurrency than those based on the other commutativity relations, they are still interesting since they are applicable to both update-in-place and deferred-update models (since $ic \subseteq c$).

⁹Recall that Σ is the set of all possible operation invocations. By $ot_i = type(i)$ and $ot_j = type(j)$ we denote the operation types of the operation invocations i and j and $\vec{in}_i = input(i)$ and $\vec{in}_j = input(j)$ denote the input parameter values of i and j (cf. Definition 7, page 52).

Table 3.6: Invocation-based Commutativity Relation for ACCOUNTS-2.

First operation (o)	Second operation (p)	$credit(d_2, a_2)$	$debit(d_2, a_2); s_2$	$transfer(d_2, e_2, a_2); s_2$	$owner(d_2); n_2$	$balance(d_2); a_2$	$total(n_2); a_2$
$credit(d_1, a_1)$		$true$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$debit(d_1, a_1); s_1$		$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$(d_1 \neq d_2 \wedge d_1 \neq e_2)$ $\vee a_1 = 0$ $\vee a_2 = 0$	$true$	$d_1 \neq d_2$ $\vee a_1 = 0$	$a_1 = 0$
$transfer(d_1, e_1, a_1); s_1$		$d_1 \neq d_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$(d_1 \neq d_2 \wedge d_1 \neq e_2)$ $\vee a_1 = 0$ $\vee a_2 = 0$	$(d_1 \neq d_2 \wedge e_1 \neq d_2)$ $\wedge d_1 \neq e_2$ $\vee a_1 = 0$ $\vee a_2 = 0$	$true$	$(d_1 \neq d_2 \wedge e_1 \neq d_2)$ $\vee a_1 = 0$	$a_1 = 0$
$owner(d_1); n_1$		$true$	$true$	$true$	$true$	$true$	$true$
$balance(d_1); a_1$		$d_1 \neq d_2$ $\vee a_2 = 0$	$d_1 \neq d_2$ $\vee a_2 = 0$	$(d_1 \neq d_2 \wedge d_1 \neq e_2)$ $\vee a_2 = 0$	$true$	$true$	$true$
$total(n_1); a_1$		$a_2 = 0$	$a_2 = 0$	$a_2 = 0$	$true$	$true$	$true$

Protocols based on ic are easier and more efficient to implement than those based on state-dependent commutativity relations. The commutativity specifications are simpler, the conflict tests are more efficient, and problems exploiting the return values in case of pessimistic protocols are omitted.

We conclude our discussions about commutativity by summarizing the relationships among the six different commutativity relations. Figure 3.4 illustrates these relationships.

PROPOSITION 7 (RELATIONSHIPS AMONG DIFFERENT COMMUTATIVITY RELATIONS)

(i) $ic \subseteq c$, (ii) $c \subseteq bc \subseteq rbc$, (iii) $c \subseteq fc \subseteq rbc$, (iv) there exist cases where $fc \not\subseteq bc \wedge fc \not\subseteq bc$, (v) in general, $lfc \not\subseteq rbc \wedge lfc \not\subseteq rbc$.

Proof (Proposition 7): (i) is already stated by Lemma 12. (ii) follows from Definition 26 and Lemma 5. (iii) follows from Definition 26 and Lemma 9. (iv) is shown by Lemma 8. (v) follows from the facts that $\overline{bc}$ is the symmetric closure of rbc (Lemma 5), $\overline{fc}$ is the symmetric closure of lfc (Lemma 9), and from (vi). This proves the proposition. $\square$

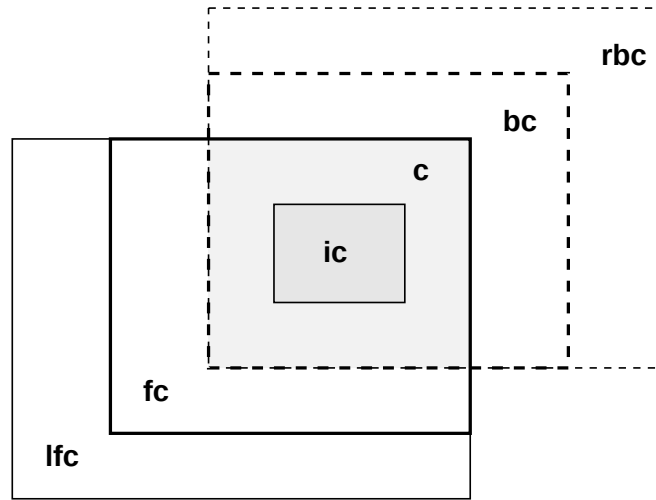


Figure 3.4: Relationships among the Different Commutativity Relations.

3.4 Cooperative Activities

In the COACT cooperative transaction model, users can cooperate if they participate in the same *cooperative activity*. To model the correct interaction, i.e., cooperation, of users involved in a cooperative activity, we introduce the concept of *cooperative activity types* (*CAT*). Every cooperative activity is an *instance* of a single cooperative activity type *CAT*.

In the following section, we formally describe how cooperative activity types are modeled in COACT. Afterwards, we present a sample *CAT* for the SEPIA hypermedia authoring scenario which was used in our demonstrator system (cf. Chapter 5). The SEPIA *CAT* is used in Section 3.5 to illustrate the history merging mechanism. In Section 3.4.3, we give a formal description of workspaces, cooperative activity instances, and the exchange meta-operations.

3.4.1 Cooperative Activity Types

In this section, we formally describe the constituents of a cooperative activity type, which is important for the understanding of the COACT model and the concepts of history merging.

DEFINITION 28 (COOPERATIVE ACTIVITY TYPE)

A *cooperative activity type* is described as a tuple $CAT = (N, O, comp, M)$ where

- N is the *name* of the cooperative activity type,
- O is a set of *invocable operation types*,
- $comp$ is a *compensation function*,
- M is a set of *merging relations*.

In the remainder of this section, the components of a cooperative activity type CAT are described in more detail.

3.4.1.1 Invocable Operations Types

The set O of a cooperative activity type $CAT = (N, O, comp, M)$ defines a finite set of invocable operation types. Actors in a cooperative activity of type CAT can invoke only operation types $ot \in O$ in their private workspaces. The invocation of these operation types is the only means to retrieve and change the state of the corresponding private workspace. The resulting instance operations appear as atomic state changes in the respective workspace and are executed, in contrast to the cooperative activity, under the control of a single actor. We assume that invocable operation types O are the same for all workspaces (or actors) in a cooperative activity¹⁰.

The exchange of operations among workspaces and the re-execution of operations in workspaces require that the specified operation types $ot \in O$ are deterministic functions and guarantee termination (cf. Definition 6). The set of all possible operations that can be invoked by actors on the workspaces, i.e., all operation instances o with $type(o) \in O$, is denoted by Ω .

3.4.1.2 Compensation Function

In COACT, we require that for every operation o a *compensation operation* o^{comp} is available. Compensation allows us to undo operations in a workspace. This may be required when conflicts occur during an exchange meta-operation or because of a user-initiated selective undo.

The compensation operation o^{comp} of an operation o semantically undoes the effects of the original operation o . It is not required that it physically restores the prior workspace state [KLS90, NZ96]. This means that the state resulting from the execution of $[o, o^{comp}]$ needs *not* to be equal to the state before o —we only require that both states are *equivalent*. Note that the operation type of a compensation operation must not necessarily be contained in the set of invocable operation types O of the CAT .

To ensure the autonomy of users within their workspaces, we compensate an operation only within a single workspace. Recall that an operation might be contained in more than one workspace history due to previous information exchanges. If an operation which occurs in more than one history should be discarded in a workspace, the operation is *not* compensated in all workspaces in which it occurs. The operation is only compensated in that particular workspace.

In COACT we make use of a *compensation function* to specify compensation operations for all operations $o \in \Omega$. The compensation function $comp$ of a cooperative activity type CAT

¹⁰This is not a restriction, since additional coordination mechanisms, like the CoCoA coordination model, can be used to further constrain the invocable operation types per user or user role, dependent on the state of a cooperative activity.

assigns to every operation instance $o \in \Omega$ its corresponding compensation operation o^{comp} , i.e., $comp(o) = o^{comp}$. The compensation function $comp$ is formally defined in Section 3.5.3.

It is the *CAT* designer's responsibility to specify the compensation function within a cooperative activity type *CAT*. Actually, the designer needs not specify compensation operations completely (which include output parameter values). It is sufficient to specify the *invocation* of the compensation operation. The input and output parameters of the operation o are used to specify the invocation i^{comp} of the compensation operation o^{comp} by means of the compensation function. The output values of the compensation operation are irrelevant. Note that the specification of the compensation operation (invocation) depends only on that part of the workspace state that is reflected in the output parameter values of the original operations. For the sake of simplicity, we assume in the following, that the information externalized by an operation is sufficient to compute the input parameters of the invocation of the compensation operation.

Example 29 (Compensation function and compensation operation) Consider the operation types of Accounts-2 (cf. Example 20 on page 64). A *debit* operation that succeeded is compensated by a corresponding *credit* operation with the same *amount* as argument. A *credit* operation is compensated by a successful *debit* operation. This can be specified by the compensation function $comp$ as follows:

$$comp(debit(account, amount):success) = \begin{cases} credit(account, amount) & \text{if } success = true \\ noop & \text{otherwise} \end{cases}$$

$$comp(credit(account, amount)) = debit(account, amount)$$

For operation $debit(account, amount):false$ the compensation operation is the *noop* operation. $Debit(account, amount):false$ does not change the workspace state, and, thus, it needs not be compensated. Assume, Alice has executed operation $o_1 = debit(a, 100):true$. If she wants to undo her operation, the compensation operation is $credit(a, 100)$ as specified by $comp$. Note that we do not just restore the account's balance to the value before the *debit* operation (state-based undo). Using operation-based undo in COACT enables us to compensate operations also when subsequent operations have been executed in the mean. For instance, if Alice has executed a second operation $o_2 = debit(a, 200):true$ it is still possible to compensate the first operation o_1 without undoing operation o_2 . $\square$

3.4.1.3 Merging Relations

The merging relations M of a cooperative activity type *CAT* are needed to allow for a semantically correct exchange of operations among workspaces. M consists of the *backward commutativity* (bc) and the *forward commutativity* relations (fc) defined over operations $o \in \Omega$.

The specification of the relations bc and fc provides for each pair of operation types $ot \in O$ a predicate involving their parameters. By applying the specified predicates to the respective operation instances, the membership in the bc or fc relation can be checked at runtime. Note

that the evaluation of the predicates depends only on those parts of the workspace state reflected in the parameters.

This specification concept corresponds to *commutativity tables*, as introduced in Section 3.3. If a pair of operation types commute backward (forward) for all possible instances, it might not be required to specify the *bc* (*fc*) predicates for this pair. The default predicate in this case is *false*, i.e., backward and forward commutativity have to be explicitly specified by the designer of a *CAT*. Alternatively, it is possible to specify the complements of the *bc* and *fc* relations as backward and forward *conflict tables*. If there is no entry for a pair of operation types contained in *O*, we assume that all pairs of operation instances of the corresponding operation types do *not* conflict, i.e., the default predicate is *false*.

The advantage of the latter approach is that if conflicts are rare, the specification overhead is decreased. This is often the case in real-world applications, where a lot of operation types operate on disjoint objects in the database. The disadvantage is that in case the designer of a *CAT* forgets to write down a predicate, it is assumed that there exists no conflict (but actually there may be a conflict).

Like the other constituents of a cooperative activity type, the merging relations are generated in the TRANSCOOP system out of a COCOA specification (cf. Sections 3.6.1 and 5.1). In COCOA, we specify conflicts among the invocable operation types of a *CAT*. The analysis tools under construction at the University of Twente [SE99, SE98a, ES98] provide verification support for the conflict relations specified in COCOA and the compensation function.

3.4.2 Sample Cooperative Activity Type for SEPIA

In order to illustrate the history merging functionality of COACT, we introduce a sample *CAT* that is used as a running example in the remainder of the thesis. The example is motivated by the cooperative hypermedia authoring system SEPIA, which has been selected as our demonstrator application.

SEPIA [SHT89] supports different activities in the authoring process, i.e., planning, content acquisition and structuring, and preparation of the final document (cf. Section 2.1.1). In contrast to traditional hierarchically structured documents, the concept of hypertext allows to have network-like document structures. SEPIA documents consist of nodes and connecting links which enable authors to create and present associative reference structures between information entities. Nodes may contain different kinds of multimedia information.

Because we study the authoring process from a database point of view, we assume a scenario in which multiple authors manipulate a collection of shared hypermedia documents, that are persistently stored in a database management system. The SEPIA hierarchical hyperdocument structure and the corresponding operations for the creation and manipulation of SEPIA document structures are modeled explicitly in the database schema [WA95, BWAH96, WA97]. Figure 3.5 shows the graphical user interface of the DBMS-based SEPIA system. All user interactions are directly executed through invocations of operation types implemented in the SEPIA database schema. For more details about the SEPIA implementation, we refer to Section 5.3.

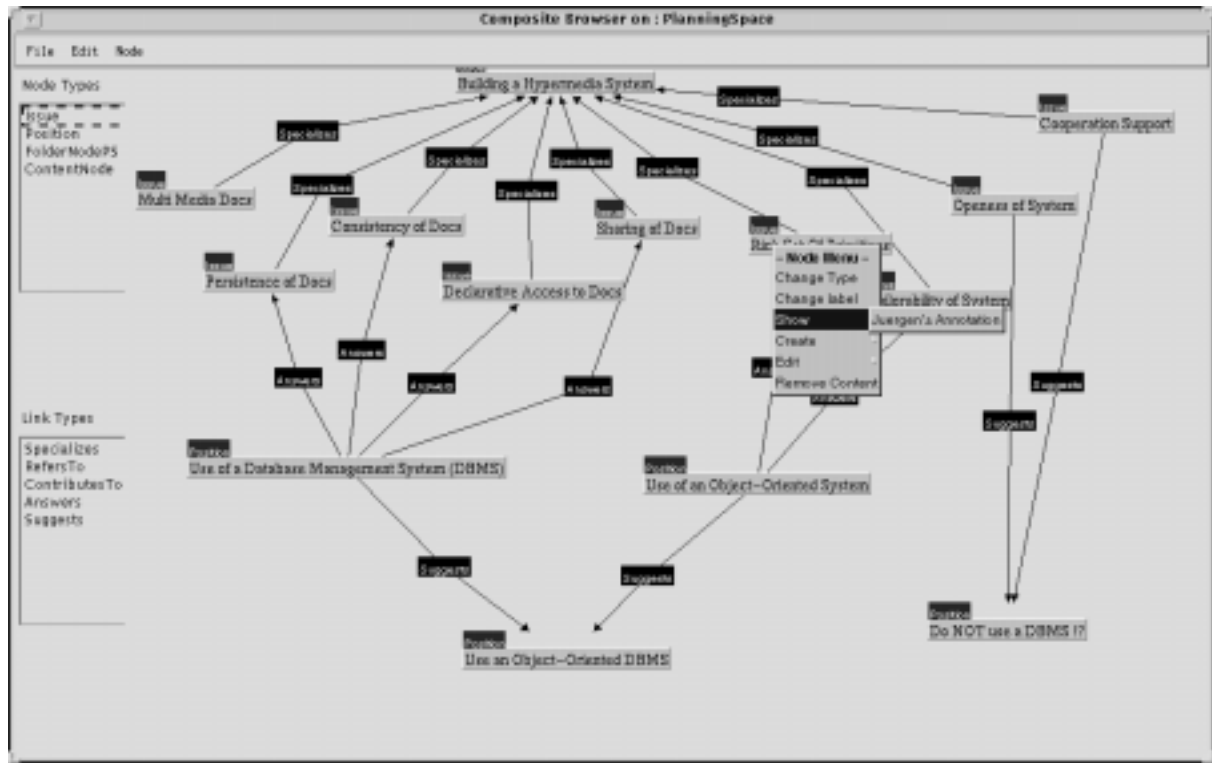


Figure 3.5: The Graphical User Interface of the DBMS-based SEPIA System.

In the following, we introduce sample invocable operation types provided by the SEPIA database schema. The presented operation types are only a subset of the invocable operation types available in the complete system. We restrict ourselves to those operation types that are needed to illustrate our concept of history merging later on.

Example 30 (Sample invocable operation types for SEPIA) For the sake of simplicity, we do not specify the following operation types formally. The signatures of the listed operation types are simplified with respect to the implementation in the SEPIA database schema.

- ***createNode(NODETYPE type, STRING name):OBJECT node*** creates a new hypermedia node in a SEPIA hyperdocument. *Type* specifies the type of hypermedia node, e.g., a *ContentNode* representing document content, *Issue* for a problem to be solved, or *Position* for an author's position. For a complete list of node types in SEPIA see [TW95a, WA95]. The object identifier of the new node is returned.
- ***setName(OBJECT node, STRING name):[STRING oldName]*** modifies the name of the specified *node* and returns the old name.
- ***getName(OBJECT node):[STRING name]*** returns the name of *node*.

- ***addContent*(OBJECT *node*, OBJECT *content*)** adds content to a node where *content* can refer to arbitrary media objects like text, image, audio, video, or a composite media object. Note that a node in SEPIA can contain an arbitrary number of content objects. The order of the content objects does not matter but duplicates are not allowed (set semantics).
 - ***removeContent*(OBJECT *node*, OBJECT *content*)** removes the specified content object from the specified node in the SEPIA document.
 - ***createLink*(LINKTYPE *type*, OBJECT *from*, OBJECT *to*):OBJECT *link*** creates a hyperlink between the specified nodes, i.e., *from* and *to*. The type of the link is determined by *type*, e.g., two content nodes can be linked by a ReferSTo link. For a complete list of link types available in SEPIA and constraints defined on the types, see [TW95a, WA95]. The object identifier of the new link is returned.
 - ***deleteNode*(OBJECT *node*):[NODETYPE *type*, STRING *name*]** deletes the specified node and returns the node type and the name of the deleted node (needed to compute compensation operations). If a link refers to *node*, the *node* is not deleted and [*null*, “”] is returned. This avoids dangling links.
 - ***deleteLink*(OBJECT *link*):[LINKTYPE *type*, OBJECT *from*, OBJECT *to*]** deletes the the *link* and returns both the end nodes of the link and the type of the link.
-

In the following, we give the compensation function and the merging relations for the SEPIA cooperative activity type.

Example 31 (Compensation function for SEPIA cooperative activity type) The compensation function for the invocable operation types in SEPIA is specified as follows:

- $comp(createNode(type, name):node) = deleteNode(node):[type, name]$
- $comp(setName(node, name):oldName) = \begin{cases} setName(node, oldName):name & \text{if } name \neq oldName \\ noop & \text{otherwise} \end{cases}$
- $comp(getName(node):name) = noop$
- $comp(addContent(node, content)) = removeContent(node, content)$
- $comp(removeContent(node, content)) = addContent(node, content)$
- $comp(createLink(type, fromNode, toNode):link) = deleteLink(link):[type, fromNode, toNode]$

- $comp(deleteNode(node):[type, name]) =$

$$\begin{cases} createNode(type, name):node & \text{if } type \neq null \\ noop & \text{otherwise} \end{cases}$$
- $comp(deleteLink(link):[type, fromNode, toNode]) =$
 $createLink(type, fromNode, toNode):link$ □

Note that for some operations compensation is not necessary (*noop*). For example, if a *setName* operation does not really change the name of a node, it needs not be compensated. The same holds for a *deleteNode* operation that does not delete the *node*. This may occur when still links are pointing to or from the node.

Example 32 (Merging relations for SEPIA cooperative activity type) The backward commutativity table describing the *bc* relation for the invocable operation types in Example 30 is given in Table 3.7. The forward commutativity table for the corresponding *fc* relation is depicted in Table 3.8. Differences to the backward commutativity relation in Table 3.7 are marked with an asterisk. □

Table 3.7: Backward Commutativity Relation for the SEPIA Cooperative Activity Type.

Second operation (<i>p</i>) First operation (<i>o</i>)	$createNode(t_2, s_2):n_2$	$setName(n_2, s_2):os_2$	$getName(n_2):s_2$	$addContent(n_2, c_2)$	$removeContent(n_2, c_2)$	$createLink(t_2, fn_2, tn_2):l_2$	$deleteNode(n_2):[t_2, s_2]$	$deleteLink(l_2):[t_2, fn_2, tn_2]$
$createNode(t_1, s_1):n_1$	<i>true</i>	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq fn_2$ $\wedge n_1 \neq tn_2$	$n_1 \neq n_2$	<i>true</i>
$setName(n_1, s_1):os_1$	$n_1 \neq n_2$	$n_1 \neq n_2 \vee$ $(os_1 = s_1$ $\wedge s_1 = s_2)$	$n_1 \neq n_2$ $\vee s_1 = s_2$	<i>true</i>	<i>true</i>	<i>true</i>	$n_1 \neq n_2$	<i>true</i>
$getName(n_1):s_1$	$n_1 \neq n_2$	$n_1 \neq n_2$ $\vee s_1 = s_2$	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$n_1 \neq n_2$	<i>true</i>
$addContent(n_1, c_1)$	$n_1 \neq n_2$	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$n_1 \neq n_2$	<i>true</i>
$removeContent(n_1, c_1)$	$n_1 \neq n_2$	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$n_1 \neq n_2$	<i>true</i>
$createLink(t_1, fn_1, tn_1):l_1$	$n_2 \neq fn_1$ $\wedge n_2 \neq tn_1$	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$n_2 \neq fn_1$ $\wedge n_2 \neq tn_1$	$l_1 \neq l_2$
$deleteNode(n_1):[t_1, s_1]$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq fn_2$ $\wedge n_1 \neq tn_2$	$n_1 \neq n_2$	$n_1 \neq fn_2$ $\wedge n_1 \neq tn_2$
$deleteLink(l_1):[t_1, fn_1, tn_1]$	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	$l_1 \neq l_2$	$n_2 \neq fn_1$ $\wedge n_2 \neq tn_1$	$l_1 \neq l_2$

Table 3.8: Forward Commutativity Relation for the SEPIA Cooperative Activity Type.

Second operation (p)								
First operation (o)	$createNode(t_2, s_2):n_2$	$setName(n_2, s_2):os_2$	$getName(n_2):s_2$	$addContent(n_2, c_2)$	$removeContent(n_2, c_2)$	$createLink(t_2, fn_2, tn_2):l_2$	$deleteNode(n_2):[t_2, s_2]$	$deleteLink(l_2):[t_2, fn_2, tn_2]$
$createNode(t_1, s_1):n_1$	$true$	$true^*$	$true^*$	$true^*$	$true^*$	$true^*$	$true^*$	$true$
$setName(n_1, s_1):os_1$	$true^*$	$n_1 \neq n_2 \vee s_1 = s_2$	$n_1 \neq n_2 \vee s_1 = s_2$	$true$	$true$	$true$	$n_1 \neq n_2$	$true$
$getName(n_1):s_1$	$true^*$	$n_1 \neq n_2 \vee s_1 = s_2$	$true$	$true$	$true$	$true$	$n_1 \neq n_2$	$true$
$addContent(n_1, c_1)$	$true^*$	$true$	$true$	$true$	$true$	$true$	$n_1 \neq n_2$	$true$
$removeContent(n_1, c_1)$	$true^*$	$true$	$true$	$true$	$true$	$true$	$n_1 \neq n_2$	$true$
$createLink(t_1, fn_1, tn_1):l_1$	$true^*$	$true$	$true$	$true$	$true$	$true$	$n_2 \neq fn_1 \wedge n_2 \neq tn_1$	$true^*$
$deleteNode(n_1):[t_1, s_1]$	$true^*$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq n_2$	$n_1 \neq fn_2 \wedge n_1 \neq tn_2$	$n_1 \neq n_2$	$true^*$
$deleteLink(l_1):[t_1, fn_1, tn_1]$	$true$	$true$	$true$	$true$	$true$	$l_1 \neq l_2$	$true^*$	$true^*$

3.4.3 Cooperative Activity Instances

A *cooperative activity instance* (for short *cooperative activity*) represents a single execution of a cooperative activity type. A cooperative activity CA is started by instantiating a cooperative activity type CAT . This results in the creation of the common workspace and the private workspaces for the participating users.

DEFINITION 29 (WORKSPACES AND WORKSPACE HISTORIES)

A *workspace* is modeled in COACT as a tuple $WS = (S_{init}, H)$ where

- S_{init} is the *initial state* of the data objects in the workspace and
- H is the *workspace history*, i.e., the history of operations executed in the workspace.

The *actual state* S_i of a workspace WS_i within a cooperative activity is the state resulting from the application of the workspace history to the initial state, i.e., $S_i = H_i(S_{init})$. To ensure that subsequent definitions are independent of S_{init} , we require that there exists a legal history H with $H(S_{\perp}) = S_{init}$. This implies that S_{init} is a consistent state (cf. Definition 18 on page 57).

Within workspaces we apply an update-in-place policy, i.e., every execution of an operation directly changes the state of the workspace where the operation was executed. The states of

other workspaces are not affected by the operation. Thus, the workspaces of a cooperative activity are isolated. Cooperation among the co-workers in a cooperative activity is established by applying one of the exchange meta-operations (described in Section 3.1).

Isolation of workspaces can be achieved (conceptually) by copying the data objects initially contained in the common workspace to all private workspaces. From these copies the users can create their own private versions of data objects that can be manipulated independently. Hence, the modifications to data objects applied by different co-workers do not interfere with each other. An alternative way of viewing the implementation is interpreting the workspace histories as intention-lists [HR83] (cf. Section 3.3).

An important assumption is that all operations invoked by an actor within a private workspace, i.e., all generated operations, are legal in the respective state. This assumption is reasonable, since we assume in COACT an interactive environment (like the SEPIA system) where the sequence of operations is interactively composed at runtime. A user selects his next action from the set of invocable operation types O . The actual input parameters of the operation invocation are supplied by the user (directly or by means of a graphical user interface). The result of the operation invocation is computed based on the actual workspace state. Hence, the resulting generated operation instance is always legal (cf. Section 3.2.3).

A cooperative activity is modeled as a set of workspace histories representing the workspaces. Each workspace in a cooperative activity has the *same* initial state S_{init} .

DEFINITION 30 (COOPERATIVE ACTIVITY)

A cooperative activity is modeled as a tuple $CA = (CAT, S_{init}, \{H_0, H_1, \dots, H_n\})$, where

- CAT is the type of the cooperative activity,
- S_{init} is the initial state in the workspaces,
- H_0 denotes the workspace history representing the common workspace in the cooperative activity,
- $H_i, i > 0$, denote the workspace histories representing the private workspaces of the users in the cooperative activity.

In practice, the number of workspaces can change dynamically as participants join or leave the cooperative activity. This is not essential for our model as long as all the workspaces have the same initial state S_{init} . To realize this flexibility, it is necessary to remember the initial state of a cooperative activity, in order to create a private workspace for new a user who joins the running cooperative activity.

The COACT model provides basic primitives to manage the execution of cooperative activities [RKT⁺95a, RKT⁺95b]. These primitives include the instantiation of a cooperative activity of a given type (*start*), the ability to suspend a user's activity to be resumed later on, and primitives to *join* or *leave* the cooperative activity. *Join* creates a new workspace for a user in an already existing cooperative activity, while *leave* discards the workspace of the user who is no

longer part of the cooperative activity. Before leaving, users will *delegate* their results to co-workers or *save* them in the common workspace (cf. definition of exchange meta-operations below).

In addition, the COACT model provides primitives for aborting or committing a cooperative activity with almost usual semantics. Those parts of a cooperative activity that are reflected in the common workspace after its commit are considered as the final result of the cooperative activity. It is assumed that all users integrate their relevant contributions into the common workspace (by means of the *save* exchange meta operation), such that there is a single result of the cooperative activity. *Commit* and *abort* discard all the private workspaces. In case of an *abort*, also the common workspace is discarded. Additional coordination mechanisms can be provided on top of COACT to constrain the set of users who are allowed to issue a commit or an abort, as well as the time of committing a cooperative activity. For example, the COCoA coordination model offers an enabling mechanism which can refer to users, user roles, and termination states of a cooperative activity (cf. Section 3.6.1).

DEFINITION 31 (PRIMITIVES FOR THE MANAGEMENT OF COOPERATIVE ACTIVITIES)

The following primitives are provided by the COACT cooperative transaction model to manage the execution of a cooperative activity.

- $Start(CAT, S_{init})$: CA instantiates the cooperative activity type CAT and creates a cooperative activity instance $CA = (CAT, S_{init}, \{H_0\})$. This results in the creation of the common workspace WS_0 with an empty workspace history $H_0 = []$ and current state $S_0 = H_0(S_{init}) = S_{init}$.
- $Join(CA)$: WS_i creates a new private workspace WS_i for cooperative activity CA with an empty workspace history $H_i = []$ and current state $S_i = H_i(S_{init}) = S_{init}$. Afterwards, $CA := (CAT, S_{init}, \mathbb{H} \cup \{H_i\})$, $H_i \notin \mathbb{H}$, where $\mathbb{H}$ denotes the set of histories in the cooperative activity CA before *join* was executed.
- $Leave(CA, WS_i)$ discards the private workspace WS_i in cooperative activity CA . Thus, $CA := (CAT, S_{init}, \mathbb{H} \setminus \{H_i\})$, where $\mathbb{H}$ denotes the set of histories in the cooperative activity CA before *leave* was executed.
- $Commit(CA)$ commits the cooperative activity CA , i.e., it discards the private workspaces $H_i \in \mathbb{H}$, $i > 0$, where $\mathbb{H}$ denotes the set of histories in the cooperative activity CA before *commit* was executed. The *result of the cooperative activity* is reflected by the common workspace with current state $S_0 = H_0(S_{init})$.
- $Abort(CA)$ simply discards all workspaces of the cooperative activity CA which ceases to exist.

Additionally, we can provide $suspend(WS_i)$ and $resume(WS_i)$ primitives. $Suspend(WS_i)$ allows a user to leave a cooperative activity without discarding the state of his private workspace

WS_i . *Resume*(WS_i) allows a user to continue his work with the state of his private workspace WS_i before *suspend* was issued. The concrete realization of the *suspend* and *resume* primitives is implementation-dependent.

Moreover, an extended *Join*(CA, WS_i) primitive, $i \geq 0$, can be defined that combines the *join*(CA) primitive (creating workspace WS_j) with a subsequent *Import*(WS_i, WS_j) exchange meta-operation (cf. Definition 32). Thus, the history of the newly created workspace WS_j is $H_j = H_i$ and, thus, the state of workspace WS_j is $S_j = H_j(S_{init})$. Note that no merge conflicts can occur, since no operations are executed in workspace WS_j before *import* is executed.

Next, we define the COACT exchange meta-operations for the exchange of operations among workspaces (cf. Section 3.1). The correctness of all exchange meta-operations is enforced by the history merging mechanism, described in the next section. History merging enforces that the resulting workspace histories remain legal (cf. Definition 17 on page 56) and, hence, it ensures the consistency of the resulting workspace states (cf. Definition 18). Merge algorithms for the implementation of the COACT exchange meta-operations are described in Section 3.5.4.

DEFINITION 32 (EXCHANGE META-OPERATIONS)

Let S_l and H_l denote the state and history of a workspace WS_l before the execution of the exchange meta-operation. Let S'_l and H'_l denote the state and history of WS_l after the execution of the exchange meta-operation. The COACT model offers the following exchange meta-operations.

- *Import*(WS_i, WS_j), $j > 0, i \neq j$, imports operations $o \in H_i$ of (a private or the common) workspace WS_i into the history H_j of the private workspace WS_j :
 - $S'_j = H'_j(S_{init})$, where $op(H'_j) \subseteq (op(H_j) \cup op(H_i))$
 - $\forall WS_l, l \neq j : S'_l = S_l, H'_l = H_l$

The *import*(WS_i, WS_j) exchange meta-operation is invoked by the owner of private workspace WS_j . This user selects a set of operations $S \subseteq H_i$ to be imported from the (private or common) workspace WS_i . Merge conflicts that may occur during *import* are resolved under the supervision of the importing user. This may result in undoing operations in workspace WS_j or in ignoring some operation in S during the merge. Thus, $op(H'_j) \subseteq (op(H_j) \cup op(H_i))$. An *import* has no effect on the workspace WS_i from which the work is being imported.

- *Delegate*(WS_i, WS_j), $j > 0, i > 0, i \neq j$, incorporates operations $o \in H_i$ of the private workspace WS_i into the history H_j of the private workspace WS_j :
 - $S'_j = H'_j(S_{init})$, where $op(H'_j) \subseteq (op(H_j) \cup op(H_i))$
 - $\forall WS_l, l \neq j : S'_l = S_l, H'_l = H_l$

$Delegate(WS_i, WS_j)$ is invoked by the owner of private workspace WS_i . This user selects a set of operations $S \subseteq H_i$ in his workspace that are then delegated to the private workspace WS_j . The owner of WS_j (the delegatee) is responsible for merging the delegated operations into his workspace and to resolve possible conflicts. The *delegate* exchange meta-operations has no effect on the workspace of the delegator.

- $Save(WS_i), i > 0$, incorporates operations $o \in H_i$ of the workspace WS_i into the history H_0 of the common workspace WS_0 :

- $S'_0 = H'_0(S_{init})$, where $op(H'_0) \subseteq (op(H_0) \cup op(H_i))$
- $\forall WS_l, l > 0 : S'_l = S_l, H'_l = H_l$

The $save(WS_i)$ exchange meta-operation is invoked by the owner of private workspace WS_i . This user selects a set of operations $S \in H_i$ in his own workspace that are then incorporated to the common workspace WS_0 . The user who invokes a *save* is responsible for the resolution of possible conflicts in the common workspace.

The main difference between *import* and *delegate* is that in the former case, the actor who initiates the exchange meta-operation is also responsible to control the merge with his private workspace, whereas in the latter case, the delegator only selects the operations and the delegatee controls the merge with his workspace. In contrast to a *delegate*, the user that initiates the *save* is also responsible for resolving possible conflicts in the target workspace (in this case the common workspace).

We conclude this section with an example of a cooperative authoring activity.

Example 33 (SEPIA cooperative authoring activity) Assume Alice and Bob want to cooperatively create a SEPIA hypermedia document. Therefore, Alice instantiates the SEPIA *CAT* starting with an empty document. For the resulting cooperative activity *CA* the common workspace is created with $S_{init} = S_\perp$ and an empty workspace history $H_0 = []$.

Alice joins the cooperative activity *CA*, which results in the creation of her private workspace **A** with $H_A = []$. To create a hypermedia node in the SEPIA document, she selects the operation type *createNode* from the invocable operation types O of the SEPIA *CAT* and invokes $createNode(Content, "CNode")$. This results in the creation of node N_1 in her workspace **A**. The workspace history is $H_A = [o_1]$ where $o_1 = createNode_1(Content, "CNode"); N_1$.

When Bob joins the cooperative activity *CA*, a private workspace **B** is created with an empty workspace history $H_B = []$. The resulting configuration of the cooperative activity is depicted in Figure 3.6. The initial state of workspace **B** is $S_\perp = S_{init}$. Since workspaces are isolated in COACT, the operation o_1 is only reflected in Alice's workspace **A**— o_1 is neither reflected in the newly created private workspace **B** nor in the common workspace.

If Bob wants to get the recent state of Alice's workspace, he has to use COACT's *import* exchange meta-operation. An alternative is to ask Alice to *delegate* some of the operations in

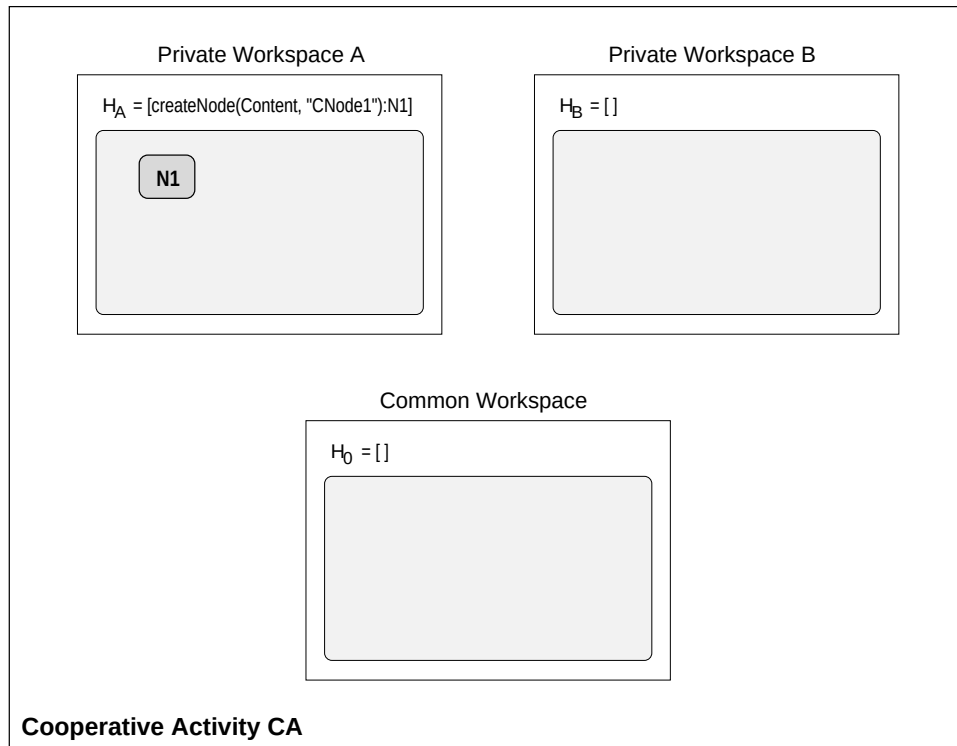


Figure 3.6: Sample SEPIA Cooperative Activity.

her private workspace to his workspace. If Alice wants to reflect the changes to her private workspace in the common workspace, she can use the *save* exchange meta-operation. $\square$

3.5 History Merging

As described in Section 3.1, history merging is the basis for all the exchange meta-operations in the COACT cooperative transaction model. In this section, we give a formal description of the history merging mechanism. In order to do this, we have to address the following issues.

1. What kind of operation sequences can be exchanged among workspaces within a cooperative activity?
2. How can we detect relevant conflicts among operations during history merging?
3. How can we resolve these conflicts?

These issues are discussed in the following. In Section 3.5.1, we utilize backward commutativity to identify consistent units of work that can be subject to an exchange of operations among workspaces. In Section 3.5.2, we present the history merge mechanism that permits the semantically correct exchange of operations among workspaces. To ensure the correctness,

we exploit the type-specific semantics of operations by means of forward commutativity to determine inter-workspace conflicts. To resolve relevant merge conflicts, we introduce a formal model of selective compensation in Section 3.5.3. In Section 3.5.4, we present a merge algorithm that can be used to implement COACT's exchange meta-operations *import*, *export*, and *delegate*. Afterwards, we discuss several refinements and optimizations of the basic merge algorithm. In Section 3.5.5, we describe how history merging can be generalized to asymmetric commutativity relations.

3.5.1 Consistent Units of Work

In this section, we introduce the notion of a *consistent unit of work* that is a central concept in COACT. An operation in a workspace history can depend on previously executed operations, i.e., its behavior is influenced by preceding operations in the history. By identifying all relevant predecessors of a given operation in the workspace history, we are able to identify consistent units of work. Only consistent units of work can be the subject of an information exchange in COACT.

Since we are interested in determining dependencies within a single workspace, we need a dependency relation that is relevant for update-in-place policy. Therefore, we apply the backward commutativity relation *bc* as described in Section 3.3.

To formalize the notion of a consistent unit of work, we introduce the concept of closed subhistories. A *closed subhistory* contains for each operation *o* in the closed subhistory every preceding operation *p* it depends on, i.e., each preceding operation *p* in the workspace history that transitively conflicts backward with *o*. A subhistory is *closed under* a set of operations *Q* if all operations of *Q* are contained in the subhistory, and each preceding operation the operations in *Q* dependent on is contained in the subhistory, too.

DEFINITION 33 (CLOSED SUBHISTORY)

A subhistory *I* of *H* is *closed* iff $\forall o \in H, p \in I : o <_H p \wedge (o, p) \notin bc \Rightarrow o \in I$. A subhistory *I* of *H* is *closed under* $Q \subseteq op(H)$ if *I* is closed and $Q \subseteq op(I)$.

We consider a closed subhistory as a consistent unit of work. We can show that such subhistories are legal if the original history was a legal one. This is an important property since it allows us to dynamically compute consistent “transactions” within a single workspace. These legal operation sequences can then be exchanged among the workspaces within a cooperative activity.

THEOREM 1

Let *I* be a closed subhistory of history *H*. If *H* is legal, then *I* is legal.

Proof (Theorem 1): Let *J* denote the restriction of *H* to operations $op(H) \setminus op(I)$. We show that the history $K = I \bullet J$ is legal. Obviously, $op(K) = op(H)$. From the definition of closed subhistories it follows that there exist no operations $o \in J, p \in I$ with $o <_H p \wedge (o, p) \notin bc$. Thus, *K* is a reordering of *H* with $\forall o, p \in H, (o, p) \notin bc : o <_K p \Leftrightarrow o <_H p$. By Proposition

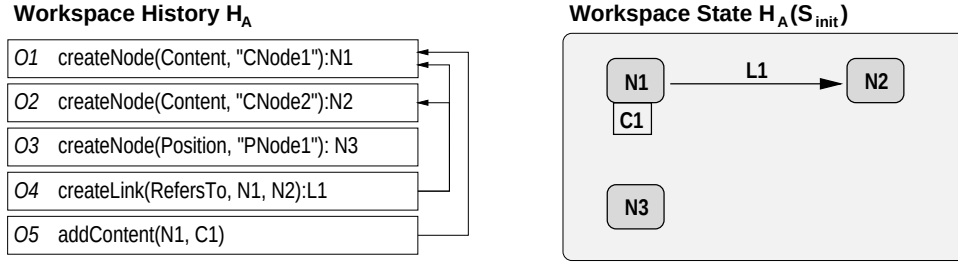


Figure 3.7: Computation of Consistent Units of Work with Backward Commutativity.

2 (page 65) it follows that K is legal and $K \equiv H$. Since K is legal, its prefix I is legal, too (Lemma 1 on page 57). This proves the theorem. $\square$

In the following example, we illustrate the concept of closed subhistories using the SEPIA cooperative activity type (cf. Section 3.4.2).

Example 34 (Closed subhistory) We continue the cooperative authoring activity that has been introduced in Example 33 (page 94). Figure 3.7 shows the history and state of Alice's workspace A. On the left side of the figure, the workspace history H_A is depicted. Each box represents an operation that has been executed by Alice who is controlling workspace A. The arrows represent the dependencies induced by the bc relation of the SEPIA CAT (cf. Table 3.7, page 89). Thus, backward conflicting operations are visualized by arrows. The right side of Figure 3.7 shows the corresponding actual workspace state $H_A(S_{init})$ with $S_{init} = S_{\perp}$.

The successful creation of the link L_1 connecting the nodes N_1 and N_2 represented by operation o_4 depends on the creation of the nodes by operations o_1 and o_2 (since a link cannot be created without the source and destination nodes). Operation o_5 which adds content to node N_1 depends on operations o_1 which created the node. In the following, we enumerate all closed subhistories of H_A .

- All prefixes of the legal history H_A are closed subhistories: $[], [o_1], [o_1, o_2], [o_1, o_2, o_3], [o_1, o_2, o_3, o_4], [o_1, o_2, o_3, o_4, o_5]$.
- Since o_3 commutes backward with o_4 and o_5 , and o_4 with o_5 , the following histories are closed subhistories of H_A : $[o_1, o_2, o_4], [o_1, o_2, o_5], [o_1, o_2, o_4, o_5]$.
- Subhistory $[o_3]$ of H_A is closed since o_3 commutes backward with all other operations in history H_A .

Assume, Alice selects operations $Q = \{o_4, o_5\} \subseteq op(H_A)$ to execute a *delegate* to Bob. There are two subhistories of H_A that are closed under Q , namely $H_1 = [o_1, o_2, o_4, o_5]$ and $H_2 = [o_1, o_2, o_3, o_4, o_5]$. $\square$

Note, there exist also other histories I with $op(I) \subseteq op(H_A)$ that are legal, e.g., $[o_2, o_1, o_5]$ and $[o_1, o_2, o_5, o_3, o_4]$. These histories are not considered as closed subhistories since they are no subhistories of H_A .

In the COACT cooperative transaction model, we consider only closed subhistories for an information exchange. An actor selects a set of operations of a workspace history which are input to the exchange meta-operations. COACT's merge algorithm (cf. Section 3.5.4) then completes the chosen operations to a subhistory that is closed under the user's selection. This ensures that only legal operation sequences are exchanged among workspaces.

To make the granularity of an information exchange as fine as possible, we prefer in COACT to exchange only those operations that are absolutely necessary to keep consistency. We call such a history *minimally closed*. For instance, in Example 34, we would prefer to exchange history $H_1 = [o_1, o_2, o_4, o_5]$ instead of $[o_1, o_2, o_3, o_4, o_5]$ since H_1 does not contain operation o_3 , which commutes backward with all other operations in H_A . Informally, a subhistory I is minimally closed under a set of selected operations Q if it contains *no* operation that commutes backward with all other operations in I and that is not part of Q .

DEFINITION 34 (MINIMALLY CLOSED SUBHISTORY)

A closed subhistory I of H is *minimally closed* under $Q \subseteq op(H)$ iff $\forall o \in I : (o \in Q) \vee (\exists p \in I : o <_I p \wedge (o, p) \notin bc)$.

The next lemma shows that there exists exactly one minimally closed subhistory for a given set of operations. This allows COACT's merge algorithm to uniquely identify the subhistory to be exchanged if a user invokes an exchange meta-operation.

LEMMA 13 Let H be a history. For each set $Q \subseteq op(H)$ there exists exactly one subhistory I of H which is minimally closed under Q .

Proof (Lemma 13): We prove the lemma by contradiction. Let H_1 be a subhistory of H that is minimally closed under $Q \subseteq op(H)$. Assume, there exists a different subhistory H_2 that is minimally closed under Q with $H_1 \neq H_2$.¹¹

We have to consider two cases: (1) $op(H_1) \subset op(H_2)$ or (2) $op(H_1) \supset op(H_2)$. In the first case, there exists an operation $o \in H_2$ and $o \notin H_1$. Since H_1 is assumed to be minimally closed under Q , Definition 34 yields that $o \notin Q$ and that there exists *no* operation p in H_1 with $o <_H p$ and $(o, p) \notin bc$. Thus, H_2 cannot be minimally closed under Q . The second case is analogous to the first. This proves the lemma. $\square$

The backward commutativity relation bc allows us to define another useful property of subhistories. We consider two subhistories of the same history to be independent if they have no common elements and there exist no backward conflicts among operations of the different subhistories.

¹¹Please recall that by definition the ordering of operations in H_1 and H_2 are the same as in H . Thus, we need only consider equality of history rather than equivalence.

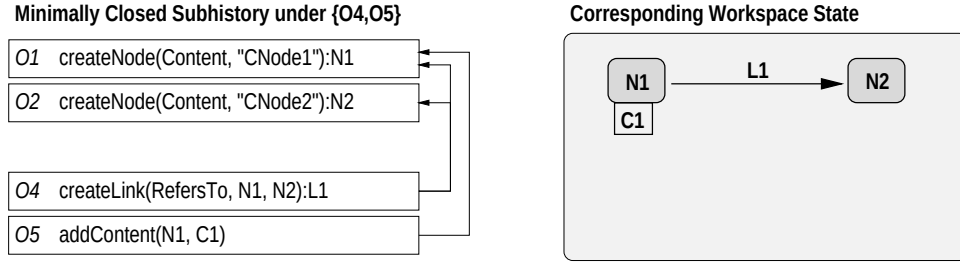


Figure 3.8: Computation of Minimally Closed Subhistory with Backward Commutativity.

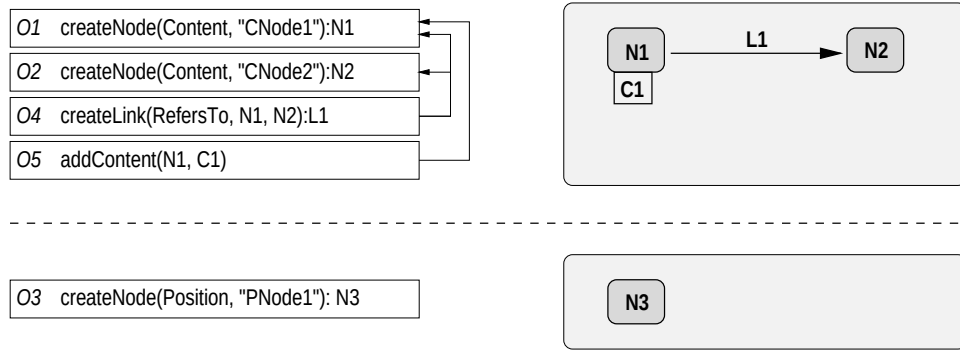


Figure 3.9: Independent Closed Subhistories.

DEFINITION 35 (INDEPENDENT SUBHISTORIES)

Two subhistories H_1, H_2 of history H are called *independent* iff $op(H_1) \cap op(H_2) = \emptyset \wedge \forall o \in H_1, p \in H_2 : (o, p) \in bc$.

The following example illustrates these newly introduced concepts using the sample SEPIA cooperative activity type.

Example 35 (Minimally closed and independent subhistories) We continue with Example 34. Assume, Alice selects operations $Q = \{o_4, o_5\}$ for an information exchange. According to Definition 34, the system computes the minimally closed subhistory under Q by selecting all backward conflicting predecessor operations of Q in the workspace history H_A . The minimally closed subhistory of H_A under Q is $[o_1, o_2, o_4, o_5]$ (see Figure 3.8). This history cannot be further partitioned into independent histories.

Assume, Alice selects operations $Q = \{o_3, o_4, o_5\}$. The minimally closed subhistory under Q is H_A . H_A can be further partitioned into two independent closed subhistories $H_1 = [o_1, o_2, o_4, o_5]$ and $H_2 = [o_3]$ (see Figure 3.9). We can exchange H_1 and H_2 independently with other workspaces because there exist no backward conflicts among operations contained in H_1 and H_2 . $\square$

Note that closed subhistories and independent subhistories are different concepts. Two independent subhistories are not necessarily closed subhistories. For instance, in the example above, the histories $[o_3]$ and $[o_4]$ are independent but $[o_4]$ is not closed. Moreover, a minimally closed subhistory may be further partitioned into independent closed subhistories as Example 35 shows. However, disjoint closed subhistories are independent, as the next lemma shows. In the COACT merge algorithm (cf. Section 3.5.4), we will always exchange independent, minimally closed subhistories separately.

LEMMA 14 Let H_1 and H_2 be two closed subhistories of H with $op(H_1) \cap op(H_2) = \emptyset$. Then H_1 and H_2 are independent subhistories of H .

Proof (Lemma 14): According to Definition 33, every operation $p \in H$ with $o \in H_1 \wedge p <_H o \wedge (p, o) \notin bc$ is included in H_1 . The same holds for H_2 . From $op(H_1) \cap op(H_2) = \emptyset$, it follows that there exist no operations $o \in H_1, p \in H_2$ with $(o, p) \notin bc$, since either $o <_H p$ or $p <_H o$. It follows that H_1 and H_2 are independent subhistories of H . $\square$

In this section, we have formalized the notion of consistent unit of work. In traditional transaction models, transactions constitute consistent units of work. Transactions are usually specified as fixed *transaction programs* which prescribe the application logic and the database operations [VWP⁺98]. Transaction programs are assumed to be programmed in a way that they keep the database consistent in terms of the application semantics (cf. Section 1.1). Especially, the resulting sequence of data operations issued during the execution of a transaction program between begin-transaction and commit are always assumed to transfer a consistent database state to another consistent state (C property in ACID). Furthermore, the resulting transactions are assumed to be atomic. This means that either all operations are executed or it appears as if none of the data operations had been executed.

In contrast to traditional transaction models, COACT is able to dynamically identify consistent units of work (or in other words, “transactions”). We assume an interactive environment where the exact operation sequence executed in the workspace is determined interactively by the user at runtime. Because it is difficult in such interactive environments to identify consistent sequences of operations beforehand, we do this at runtime exploiting the operation semantics captured by the *bc* relation. A “transaction”, that can be exchanged among workspaces, consists of all selected operations plus all relevant predecessor operations and only those (minimally closed subhistory). Note that it is not required that a consistent unit of work corresponds to the directly preceding operations. This is a major difference to the traditional transaction model.

3.5.2 Correct Merging of Histories

In this section, we introduce the notion of correct operation exchanges among different workspaces within a cooperative activity. We will call two histories *correctly merged* if the resulting history preserves all *intra*-workspace dependencies and there exist no relevant *inter*-workspace conflicts. We show that merged histories that obey our criteria are always legal. Thus, by using

the notion of correctly merged histories it is possible to establish cooperation among the users in a cooperative activity without affecting consistency of their workspaces, i.e., the legality of the respective workspace histories.

We have used the *bc* relation to describe intra-workspace dependencies and to identify consistent units of work that can be subject to an information exchange. If such a unit of work, represented by a minimally closed subhistory, is incorporated into another workspace history, we have to ensure that the exchanged operations behave the same in the target history as in the source history. Since we are here interested in determining conflicts (or dependencies) between operations executed in *different* workspaces, we need a dependency relation that is relevant for deferred-update policy. Therefore, we apply the forward commutativity relation as described in Section 3.3.2.1.

Two histories can only be merged if they are based on the same initial state and both are legal. This is captured by the next definition.

DEFINITION 36 (MERGEABLE HISTORIES)

We call two workspace histories H_1 and H_2 *mergeable* iff (1) both histories are based on the same initial (consistent) state S_{init} and (2) H_1 and H_2 are both legal in state S_{init} .

The first condition is necessary, since the definition of forward commutativity and the resulting Proposition 5 are based on the equality of the initial states (represented by equal histories). The second condition is required, since the history merging mechanism assumes that the source histories are already legal in order to construct a new legal history containing operations from both source histories.

A *merged history* M is a history that consists of operations that originate from different workspace histories. Note that we do not constrain the order of operations in M to comply with the execution orders in the source histories. Among all possible merged histories we consider only those as *correct* that fulfill the conditions of a correct merge (cf. Definition 38).

DEFINITION 37 (MERGED HISTORY)

Let H_1 and H_2 be two mergeable histories. A *merged history* M consists of operations $op(M) \subseteq op(H_1) \cup op(H_2)$.

It is not required that the two histories to be merged are disjoint, i.e., it need *not* hold that $op(H_1) \cap op(H_2) = \emptyset$. As a result of a previous information exchange an operation can occur in both source histories. The occurrence of identical operations in both source histories enables us to ignore some of the forward conflicts that may occur between operations of different histories, as we will see in Definition 38. Recall that identity of operations is determined by their associated operation identifier (cf. Section 3.2.2). We assume that the operation identifiers assigned to operations at generation time are unique within all workspace histories in a cooperative history. Operations that are re-executed keep their original operation identifier.

Example 36 (Merged histories) Consider a SEPIA cooperative activity that was started with an empty hyperdocument. This empty document represents the initial state of the private

workspaces of Alice and Bob (see Figure 3.10). Assume that both authors have imported a single operation q_1 from co-author Charly. The *re-execution* of operation q_1 created node N_1 in both workspaces. After that, Alice creates a node N_2 (operation o_1) and a link L_1 between N_1 and N_2 (operation o_2). Bob creates a node N_3 by executing operation p_1 . After that, he deletes node N_1 (operation o_2). Figure 3.10 shows the resulting workspace histories H_A and H_B and the corresponding workspace states.

Now a merged history M is constructed from the operations in $op(H_A) \cup op(H_B) = \{q_1, o_1, o_2, p_1, p_2\}$. The order of the operations $<_M$ is not constrained. In the following, we discuss the correctness of sample merged histories constructed from H_A and H_B .

- The empty history $[]$ is a merged history. The empty history is always legal.
- H_A and H_B are merged histories. They are legal since the source histories are legal by assumption. Also, all prefixes of H_A and H_B are legal merged histories.
- $M_1 = [q_1, o_1, o_2, p_1]$ is legal since operation p_1 is not influenced by the other operations in M_1 (p_1 commutes forward with the other operations in M_1 !). Thus, p_1 behaves the same in M_1 as in H_B .
- $M_2 = [q_1, o_1, o_2, p_1, p_2]$ is *not* legal since p_2 is not legal. In its original execution it returned $[Content, "CNode1"]$ which indicates a successful deletion of node N_1 . In M_2 , p_2 would return $[null, ""]$ since it is not possible to delete node N_1 because of link L_1 . Note, p_2 does *not* commute forward with operation o_2 that created link L_1 .
- In contrast to M_2 , $M_3 = [q_1, o_1, p_1, p_2]$ is legal since the operation that created link L_1 is omitted in M_3 . Note that p_2 conflicts forward with operation q_1 but q_1 already appeared before p_1 in history H_B ($q_1 \in view(p_1)$!). In this case, we can ignore the forward conflict since the execution of p_2 is based on a workspace state where the effects of q_1 are already reflected.
- $M_4 = [q_1, o_1, p_2, p_1]$ is legal. Note that M_4 does not preserve the original order $p_1 <_{H_B} p_2$ in history H_B . This does not matter since p_1 and p_2 are independent, i.e., commute backward.
- $M_5 = [q_1, o_1, o_2, p_2]$ is legal but $M_6 = [q_1, o_2, o_1, p_2]$ is not because the order of o_2 and o_1 is changed with respect to H_B and the operations do not commute backward.
- $M_7 = [q_1, p_1, p_2, o_1]$ is legal but $M_8 = [q_1, p_1, p_2, o_1, o_2]$ is not legal because the creation of a link (operation o_2) requires the existence of the source and destination nodes.

There exist a lot of other merged histories which are not legal, e.g., $[p_2]$ and $[o_2]$, and all their continuations. $\square$

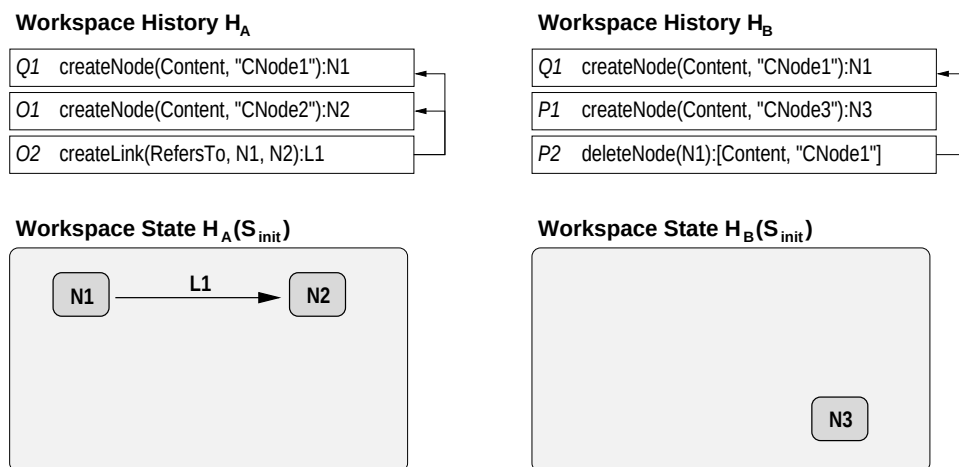


Figure 3.10: Example for Merged Histories.

Example 36 provided an initial intuition when merged histories are correct, i.e., legal. Informally, we consider two histories *correctly merged* if the resulting merged history obeys all *intra-workspace dependencies* (induced by $\overline{bc}$) and there exist no *relevant inter-workspace conflicts* ($\overline{fc}$).

TENTATIVE DEFINITION (CORRECTLY MERGED HISTORY)

A merged history M , constructed from two mergeable histories H_1 and H_2 , is correctly merged iff

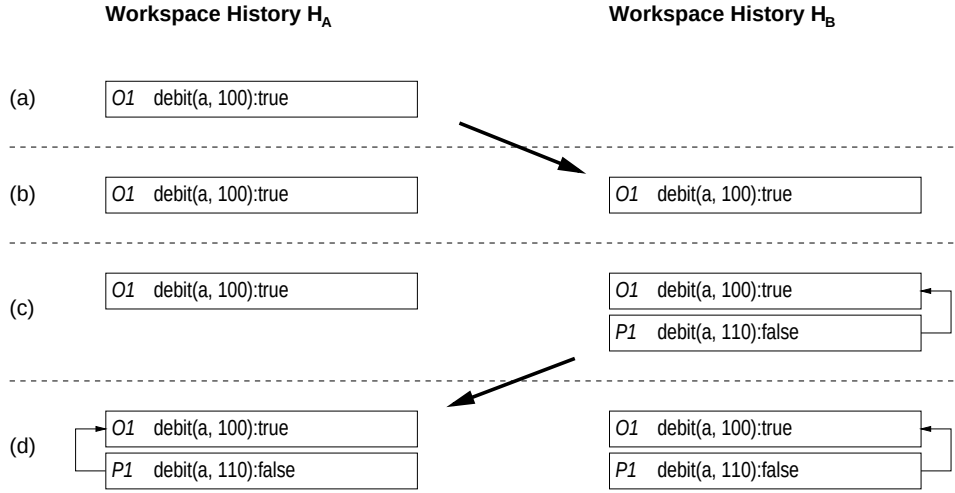
- (1) the restrictions of M to the operations in H_i are closed subhistories of H_i , $i = 1, 2$, and
- (2) all operations in $op(H_1) \setminus op(H_2)$ and $op(H_2) \setminus op(H_1)$ pairwise commute forward.

Condition (1) ensures that no operation turns illegal as a result of reordering backwardly conflicting operations of H_i in M , $i = 1, 2$. Condition (2) ensures that no operation turns illegal as a result of including forwardly conflicting operations from different source histories.

Note, that we do not check for forward conflicts among operation pairs where (at least) one of the two operations appears in both source histories. The reason is that either

1. one of the operations (e.g., operation o) was already present in history H when the second operation p was generated in H (update-in-place situation: $o \in view(p)$), or
2. both operations were generated in different workspaces (deferred-update situation: $o \notin view(p) \wedge p \notin view(o)$).

The first case corresponds to an update-in-place policy and, hence, backward commutativity applies instead of forward commutativity. In this case, Condition (1) in our tentative definition ensures that both operations are ordered properly in M if they conflict backward. In the second case, we assume that the source histories H_1 and H_2 have already been correctly merged and,

Figure 3.11: Incremental Merge with $bc \neq fc$.

hence, fulfill the two conditions of our tentative definition. Therefore, we can ignore forward conflicts for such operation pairs.

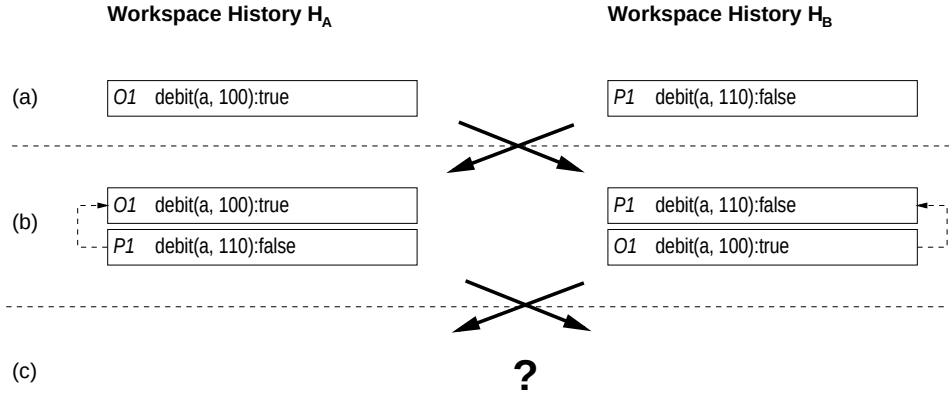
Unfortunately, the subtle difference between the bc and fc relation causes a problem with our tentative definition of correctly merged histories. This problem is illustrated by the next two examples.

Example 37 (Incremental merge with $bc \neq fc$) Consider the ACCOUNTS-2 sample operation types of Example 20. Alice has executed the operation $o_1 = \text{debit}(a, 100):\text{true}$ (cf. Figure 3.11(a)). Bob *imports* operation o_1 . This is possible since his private workspace is empty. His merged workspace history is now $[o_1]$ as shown in Figure 3.11(b). Afterwards, Bob invokes operation $p_1 = \text{debit}(a, 110)$ which returns *false*. The resulting backward conflict dependency among p_1 and o_1 is depicted by a thin arrow in Figure 3.11(c). Now, Alice merges workspace history H_B with her workspace history. The resulting merged history $[o_1, p_1]$ is legal and fulfills the tentative definition of a correct merge. $\square$

Example 38 (Merge anomaly resulting from $bc \neq fc$) Assume, Alice and Bob *concurrently* execute $o_1 = \text{debit}(a, 100):\text{true}$ and $p_1 = \text{debit}(a, 110):\text{false}$ in their workspaces (cf. Figure 3.12(a)). If either actor imports the workspace history of the other one, this may result in two different merged histories as shown in Figure 3.12(b). Both merged histories fulfill the conditions of our tentative definition of correct merges.

If these histories are merged again, there does not exist a merged history that fulfills the tentative definition of correct merges. The reason is that $o_1 <_{H_A} p_1 \wedge (o_1, p_1) \notin bc$ and $p_1 <_{H_B} o_1 \wedge (p_1, o_1) \notin bc$. No merged history can reconcile the contradictory orderings. $\square$

This anomaly results from the fact that $(o_1, p_1) \in fc$ but $(o_1, p_1) \notin bc$. Because $(o_1, p_1) \in fc$, a merge of $[o_1]$ with $[p_1]$ is possible with both orders $[o_1, p_1]$ and $[p_1, o_1]$ as Example 38

Figure 3.12: Anomaly resulting from $bc \neq fc$.

shows. If these histories are merged again, it is assumed that p_1 depends on o_1 and vice versa. But this dependency is artificial if both operations were generated in an deferred-update relationship, i.e., if $o_1 \notin \text{view}(p_1) \wedge p_1 \notin \text{view}(o_1)$. The artificial backward conflict dependencies are depicted as dashed arrows in Figure 3.12(b).

There are two possible approaches to solve this problem in the definition of correctly merged histories:

- We can use the model-independent commutativity relation $c = fc \cap bc$ (cf. Section 3.3.3) for both computing intra-workspace dependencies and inter-workspace dependencies. This solves the problem but decreases the possible degree of cooperation since c is more restrictive than bc and fc . For instance, in Example 38, a merge of $[o_1]$ with $[p_1]$ is then no longer possible since $(o_1, p_1) \notin c$, whereas the incremental merge in Example 37 is allowed.
- Alternatively, we can ignore backward conflicts among operations in a single history when both operations are not in the other operation's view (cf. Definition 23 in Section 3.3.1.3). The merge of $[o_1, p_1]$ and $[p_1, o_1]$ in Example 38 is then possible. By considering the views of operations we can determine if the initial executions of operations are in an update-in-place relationship or in a deferred-update relationship. Thus, we can decide if fc or bc has to be applied between two operations.

Despite the fact that anomalies resulting from $bc \neq fc$ appear to occur rather infrequently and the previous example appears rather artificial and harmless (“why should we include an operation that does fail and does not change the workspace state?”), we have chosen to follow the second approach. We want to exploit the greater expressiveness of different relations for describing intra-workspace dependencies and inter-workspace conflicts. Moreover, such a definition of correctly merged histories can then be extended to asymmetric commutativity

relations rbc and lfc (see Section 3.5.5). Nevertheless, it remains possible to use the c relation instead of bc and fc in the following definition. This does not touch the correctness since “stronger” commutativity relations always can be used (cf. Proposition 7, page 82).

The next definition introduces the notion of correctly merged histories we use in the CoACT cooperative transaction model.

DEFINITION 38 (CORRECTLY MERGED HISTORY)

A legal history $H = [o_1, \dots, o_n]$, where $o_i \in view(o_{i+1}), i = 1, \dots, n \Leftrightarrow 1$, is a *correctly merged history*.

A merged history M , constructed from two mergeable and correctly merged histories H_1 and H_2 , is a *correctly merged history* iff

1. $\forall o, p \in H_i, i = 1, 2 : (p \in M \wedge o <_{H_i} p \wedge (o, p) \notin bc \wedge o \in view(p)) \Rightarrow o \in M$
2. $\forall o, p \in H_i, i = 1, 2 : (o <_{H_i} p \wedge (o, p) \notin bc \wedge o \in view(p) \wedge o, p \in M) \Rightarrow o <_M p$
3. $\forall o \in (op(H_i) \setminus op(H_j)), p \in (op(H_j) \setminus op(H_i)), i \neq j : (o, p) \notin fc \Rightarrow o \notin M \vee p \notin M$

The first part of the definition states that a legal history is considered as a correctly merged history if all operations are in an update-in-place relationship and occur in the order they were generated. Such a history corresponds to a situation where only the workspace owner has invoked operations in his workspace (and no merge occurred before).

A correctly merged history, resulting from a merge of two other correctly merged histories, has to obey the following conditions.

Condition 1 expresses that all *relevant predecessor operations* of any operation that is part of the merged history M have to be included in M , too. Thus, if an operation p is included in M , every preceding operation o that does not commute backward has also to be included in M , but *only* if o is in the view of p . If this is not the case, the operation p does not depend on the o because in p 's initial execution, o was not reflected in the workspace (history). If $o \notin view(p)$ and $p \notin view(o)$ holds, the operations were initially invoked in different workspaces under a deferred-update policy. It follows, that backward conflicts can be ignored. If $o \notin view(p)$ but $p \in view(o)$ holds, then $(o, p) \in bc$, since it is assumed that the source histories are legal.

Condition 2 requires that a *relevant ordering* of operations in a source history is preserved in the merged history M . This means that operations from a source history that do not commute backward with each other have to appear in the merged history in the same order as in the source history, but, again *only* if the preceding operation is in the view of the succeeding operation.

Condition 3 states, that if two operations o and p are not contained in both source histories and do not commute forward, then only one of them can be included in the merged history M . This means that only if neither o nor p is already contained in both source histories, we have to

consider a forward conflict between o and p (*relevant conflicts*). If either o or p is contained in both histories, we can ignore a forward conflict for the following reasons¹².

1. $o \in \bigcap op(H_i) \wedge o <_{H_k} p \wedge o \in \text{view}(p) \wedge p \notin \text{view}(o)$: The operation o contained in both histories had already been present in the workspace at the time p was generated. Thus, update-in-place policy applies in this case, and only backward conflicts have to be considered, but no forward conflicts.
2. $o \in \bigcap op(H_i) \wedge o <_{H_k} p \wedge o \notin \text{view}(p) \wedge p \notin \text{view}(o)$: The operation o contained in both histories had not been present in the workspace when operation p was generated. The presence of both o and p in H_k can result only from a previous merge. There, forward commutativity of o and p has already been holding, since it is assumed that the source histories are legal. Hence, we do not have to test again if o and p do commute forward.
3. $o \in \bigcap op(H_i) \wedge o <_{H_k} p \wedge o \notin \text{view}(p) \wedge p \in \text{view}(o)$: In this case, o was generated in a workspace where operation p already had been present, in the order $p < o$. Thus, update-in-place policy applies and only backward conflicts have to be considered, but no forward conflicts. Since the operations appear in a different order in H_k , $(o, p) \in bc$ must hold because the source histories are assumed to be legal.

Next, we illustrate how the correct merging of histories works. The following example continues Example 36.

Example 39 (Correctly merged history) The upper part of Figure 3.13 shows the workspace histories of Alice and Bob. The arrows within a single workspace show the intra-workspace dependencies. The “flash” arrows between different workspace histories indicate forward conflicts among operations.

Assume, Alice wants to *import* all operations from Bob's workspace. Operation q_1 is already reflected in her workspace. Operation p_1 does not depend on any other operation in H_B and does not forward conflict with any operation in H_A . Thus, p_1 can be included in the merged history M_A since the conditions of Definition 38 are satisfied. If Alice wants to include p_2 , conditions 1 and 2 require that also operation q_1 is included in the merged history before p_2 , since operation p_2 depends on operation q_1 in history H_B . There exist two forward conflicts for p_2 . The first conflict with q_1 can be ignored because q_1 is contained in both histories H_A and H_B (Condition 2). The second conflict with operation o_3 cannot be ignored. Condition 3 implies that only one of p_2 and o_2 can be included in a merged history.

The left side of Figure 3.13 shows a correctly merged history M_B where p_2 is not included. The right part of Figure 3.13 shows a correctly merged history M_B that, for example, resulted

¹²The following enumeration is an extensive case analysis. If either o or p is contained in both source histories, there exists a history H_k with $o \in H_k \wedge p \in H_k$. Without loss of generality, we assume that $o <_{H_k} p$. Then, either (1) $o \in \text{view}(p) \wedge p \notin \text{view}(o)$, (2) $o \notin \text{view}(p) \wedge p \notin \text{view}(o)$, or (3) $o \notin \text{view}(p) \wedge p \in \text{view}(o)$. The case $o \in \text{view}(p) \wedge p \in \text{view}(o)$ does not exist.

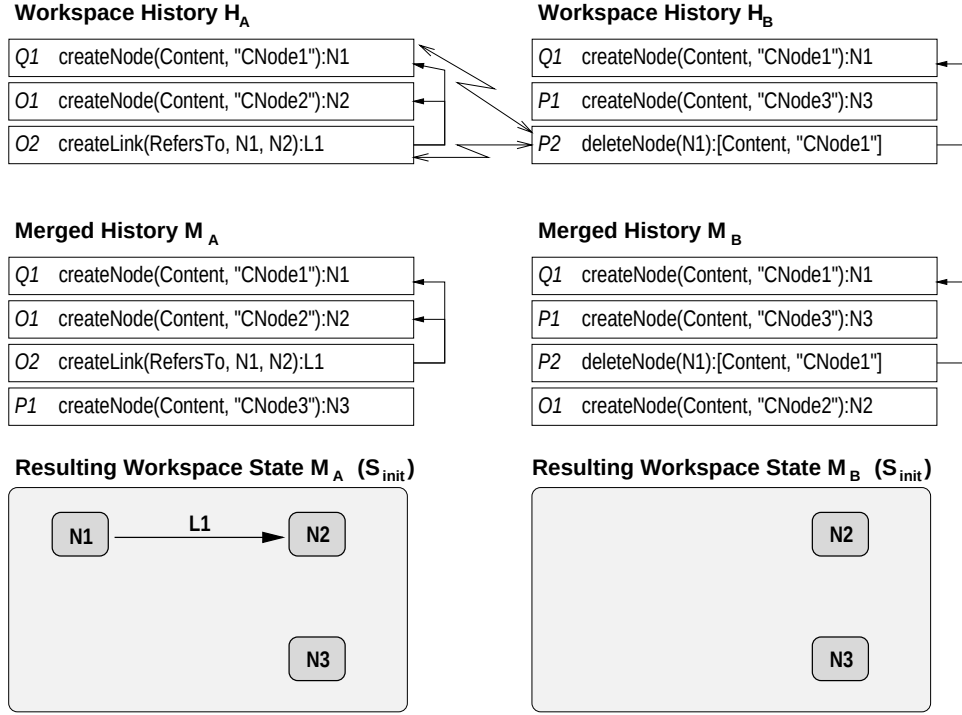


Figure 3.13: Example of Correctly Merged Histories.

from a *delegate* exchange meta-operation from Alice to Bob. There, operation p_2 is included, instead of o_2 . $\square$

In the following, we prove that correctly merged histories are legal histories.

LEMMA 15 Let H be a correctly merged history. Suppose I is a reordering of H with $\forall o, p \in H, (o, p) \notin bc, o \in view(p) : o <_I p \Leftrightarrow o <_H p$. If H is legal, then I is legal and $H \equiv I$.

Proof (Lemma 15): The history I can be obtained from history H by a finite sequence of reordering steps. The proof proceeds by induction on the number of reordering steps. If there are no such steps, then $H = I$ and the result is immediate.

Suppose the result is true for any number of reordering steps up to $k \geq 1$, and suppose that I is obtained from H by $k + 1$ reordering steps. Then there exists another history J such that J is obtained from H in k reordering steps and I is obtained from J in one reordering step. The induction hypothesis implies that J is correctly merged and legal and that $J \equiv H$.

Then, there exist operations o and p and histories K_1 and K_2 such that $K_1 \bullet o \bullet p \bullet K_2 = J$ and $K_1 \bullet p \bullet o \bullet K_2 = I$, and either (1) $(o, p) \in bc$ or (2) $(o, p) \notin bc \wedge o \notin view(p)$.

(1) If o and p commute backward, $I \bullet p \bullet o$ is legal and $K_1 \bullet o \bullet p \equiv K_1 \bullet p \bullet o$.

(2a) If $(o, p) \notin bc \wedge o \notin view(p) \wedge p \notin view(o)$, o and p are in a deferred-update relationship. This means that o and p were initially executed in different workspaces and are contained in H due to a previous correct merge. From the definition of a correctly merged

history (Condition 3) follows that $(o, p) \in fc$. Let $K_1 \bullet o \bullet p$ result from a merge of $K_1 \bullet o$ and $K_1 \bullet p$. Both $K_1 \bullet o$ and $K_1 \bullet p$ must be legal, since the definition of correctly merged histories requires that the source histories are mergable, and, hence, legal (cf. Definition 36). Since o and p commute forward, it follows that $K_1 \bullet p \bullet o$ is legal and $K_1 \bullet o \bullet p \equiv K_1 \bullet p \bullet o$.

(2b) The case where $(o, p) \notin bc \wedge o \notin view(p) \wedge p \in view(o)$ does not exist since this is not allowed by a correct merge (Condition 2).

Since $K_1 \bullet o \bullet p \bullet K_2$ is legal, Proposition 1 implies that $K_1 \bullet p \bullet o \bullet K_2$ is legal, too, and $K_1 \bullet o \bullet p \bullet K_2 \equiv K_1 \bullet p \bullet o \bullet K_2$. This yields the lemma. $\square$

The following theorem states that merged histories satisfying the conditions of Definition 38 are always legal.

THEOREM 2

A correctly merged history is legal. Moreover, any two correctly merged histories M and N with $op(M) = op(N)$ are equivalent.

Proof (Theorem 2): Assume, the correctly merged history M was constructed from the mergable and correctly merged histories H_i , $i = 1, 2$. Let $O_\cap = (op(M) \cap \bigcap op(H_i))$. Let K denote the restriction of M to operations in $O_\cap$, I_i denote the restriction of H_i to operations in $op(M) \setminus O_\cap$, and J_i denotes the restriction of H_i to operations in $op(H_i) \setminus op(M)$. It follows that $op(M) = O_\cap \cup \bigcup op(I_i)$, since operations in J_i are not contained in the merged history M .

The idea of the proof is as follows. (1) We show that the reordering $L_i = K \bullet I_i \bullet J_i$ of H_i is legal, $H_i \equiv L_i$, and $K \bullet I_i$ legal. (2) Afterwards, we prove that all histories $M' = K \bullet G$ are legal and equivalent, where $op(G) = op(M) \setminus O_\cap$ and $<_G \supseteq \bigcup (<_{I_i} \cap \overline{bc})$. Note that M is a reordering of M' . (3) The last step is to show that the correctly merged history M (and any other correctly merged history N with $op(M) = op(N)$) is legal and equivalent to an $M' = K \bullet G$. This proves the theorem.

(1) For all operations that appear in a different order in H_i and L_i , i.e., $o <_{H_i} p \wedge p <_{L_i} o$, it is ensured by Conditions 1 and 2 in Definition 38 that $(o, p) \in bc \vee o \notin view(p)$. Thus, by Lemma 15 it follows that L_i is legal and $L_i \equiv H_i$. The prefix of $K \bullet I_i$ of L_i is legal, too.

(2) Since $K \bullet I_i$, $i = 1, 2$, is legal and all operations $o \in I_i$ commute forward with all operations $p \in I_j$, $i \neq j$ (Condition 3 in Definition 38), Proposition 5 yields that all histories $M' = K \bullet G$ are legal and equivalent.

(3) It is easy to see that $op(M') = op(M)$. Condition 1 in Definition 38 ensures that for all operations $o <_M p \wedge p <_{M'} o$ it holds that $(o, p) \in bc \vee o \notin view(p)$. By Lemma 15 it follows that M is legal and $M \equiv M'$. The same holds for any correctly merged history N with $op(M) = op(N)$. $\square$

We now show that correctly merged histories need not be serializable with respect to operations generated by different users in their private workspaces. Unfortunately, it is not straightforward to establish a formal relationship between the traditional ACID transaction model and the COACT cooperative transaction model, since COACT has *no* concept of an atomic transaction. There is nothing like a begin-transaction and commit-transaction event in a workspace

history that marks an atomic, consistent, and isolated unit of work. As discussed in Section 3.5.1, consistent units of work are dynamically computed in COACT for the purpose of an information exchange.

To that end, we introduce an artificial concept of “transactions” for COACT. Since a cooperative activity is long running and open-ended and the workspaces may diverge over time, but need not be disjoint, we need to define the “transactions” for each workspace history separately. We use the following characterization of “transactions”, i.e., personal work units, for a correctly merged workspace history M . Assume that in a cooperative activity CA users $U_i, i = 1, \dots, n$, participate. With WS_i we denote the private workspaces of user U_i . Let T_i denote the restriction of M to operations generated by user U_i in the respective workspace WS_i . We consider T_i as a “transaction” in history M . Now, we are able to apply the definition of (conflict-)serializability of histories (cf. in Section 3.3.1.2) in COACT.

The following example shows that correctly merged histories need not be serializable with respect to our notion of transactions as personal work units in workspace histories. In Section 3.5.4.2, we give more complex examples of non-serializable, but correctly merged histories (in the context of the SEPIA cooperative activity).

Example 40 (Non-serializable correctly merged history) Reconsider the situation in Figure 3.11(d) on page 104. Suppose, Alice executes operation $o_2 = \text{debit}(a, 5):true$ in her workspace. Bob then merges $H_A = [o_1, p_1, o_2]$ with his own history $[o_1, p_1]$. His new history is then, e.g., $[o_1, p_1, o_2]$. Suppose, Bob executes operation $p_2 = \text{debit}(a, 3):false$. Thus, H_B is then $[o_1, p_1, o_2, p_2]$. All those histories are correctly merged.

$T_A = [o_1, o_2]$ and $T_B = [p_1]$ are considered as “transactions” in H_A . However, H_A is not serializable since $o_1 <_{H_A} p_1 \wedge (o_1, p_1) \notin bc$ and $p_1 <_{H_A} o_2 \wedge (p_1, o_2) \notin bc$. The serialization graph for H looks like $T_A \rightarrow T_B \rightarrow T_A$. Hence, H_A is not (conflict-)serializable.

$T_A = [o_1, o_2]$ and $T_B = [p_1, p_2]$ are considered as “transactions” in H_B . The serialization graph for H_B looks like $T_A \rightarrow T_B \rightarrow T_A \rightarrow T_B$ since $o_2 <_{H_B} p_2 \wedge (o_2, p_2) \notin bc$. Thus, H_B is not (conflict-)serializable. $\square$

This shows that our history merging mechanism is able to produce *non*-serializable, but still semantically correct, i.e., legal histories. Thus, the isolation property of traditional transaction models is relaxed in a controlled way, while consistency of the workspace states within a cooperative activity is guaranteed in any case.

Before we describe the basic COACT algorithm for correctly merging histories, we introduce the notion of *compensation* in the next section.

3.5.3 Compensation

Compensation allows us to undo operations in a workspace history. This may be required in order to resolve merge conflicts caused by Condition 3 in the definition of correctly merged histories (cf. Definition 38 on page 106). Additionally, compensation allows for the interactive

exploration of different problem solutions carried out within the private workspaces. Using selective compensation, a user can retract his previous decisions, i.e., selectively undo operations generated or re-executed in his private workspace. This is especially important for the interactive processing in cooperative activities [PK92, PK94, CD95].

Compensation is a very common approach to realize undo-behavior for long-running transactions [Gra81, GMS87, Elm92, DDS97]. Although the mechanism of compensation is used in almost every advanced transaction model (e.g., [GMS87, ELLR90, GMGK⁺91, Wei91, WS92, DHL91, WR92, BOH⁺92, Ley95, CD96, KRC⁺96, RSS97, DDS97]), compensation is introduced in an informal way in these models. In this section, we present a *formal model of selective compensation* in the context of COACT.

The first attempt to formalize compensation was presented in [KLS90], which tried to unify rollback and compensation. In contrast to a rollback, a compensation transaction undoes the effects of a transaction rather than physically restoring the database state (using recovery algorithms). The resulting notion of compensation in [KLS90] was very restrictive in terms of what compensation transactions have to guarantee [RSS97]: the compensation transaction, directly executed after the original transaction, has to generate a state of the accessed data objects that is *equal* to the state at the point in time the original transaction started.

In COACT, we extend the approach of [KLS90] using state equivalence instead of state equality in the definition of compensation operations o^{comp} . The state resulting from the execution of $[o, o^{comp}]$ need *not* be *equal* to the database state before o —we only require that both states are *equivalent* [KTW96c, KTW96b, NZ96, RSS97]. This implies that the compensation operation o^{comp} of an operation o is semantically undoing the effects of the original operation o rather than physically restoring the prior workspace state.

DEFINITION 39 (COMPENSATION OPERATION)

The operation o^{comp} is a *compensation operation* for operation $o \in \Omega$ iff the following holds:
 $\forall$ histories $H : H \bullet o \text{ legal} \Rightarrow (H \bullet [o, o^{comp}] \text{ is legal} \wedge H \equiv H \bullet [o, o^{comp}])$

Definition 39 states that the execution of o^{comp} immediately after o is simply the logical undo of o , i.e., no subsequent operation can observe that the sequence $[o, o^{comp}]$ is part of the history. Note that Definition 39 requires that H and $H \bullet [o, o^{comp}]$ are equivalent for *all* histories H where $H \bullet o$ is legal. This means that the definition of compensation operations is *state-independent* since $H(S_{\perp})$ can be an arbitrary consistent state. This constrains the construction of compensation operations, as we will see in the following example.

Example 41 (Compensation operation) Consider a $debit(a, 100):true$ operation in Accounts-2 example on page 64. Suppose, the balance of account a is 200 in the state S immediately before the execution of the $debit$ operation, i.e., $\Pi_2(S(a)) = 200$.

A compensation operation for $debit$ is, e.g., $credit(a, 100)$ (cf. Example 29, page 85). Another candidate for a compensation operation might be a $setBalance(a, 200)$ operation. The compensation operation $credit(a, 100)$ can be used in *every* workspace state S' where $debit(a, 100):true$ is legal. The operation $setBalance(a, 200)$ can only be used in states S''

where $\Pi_2(S''(a)) = 200$ to compensate the $\text{debit}(a, 100):true$. Thus, according to Definition 39 $\text{setBalance}(a, 200)$ is not a compensation operation for $\text{debit}(a, 100):true$. $\square$

Compensation becomes more difficult if we want to compensate an operation o in a history in case further operations have been performed. The approach used in SAGAs [GMS87], the multi-level transaction model [Wei91, WS92], and the open-nested transaction model [WS92, MRW⁺93] is to compensate o and all successor operations in *inverse order* (cf. Section 2.3.2). This conservative compensation policy is captured by the next lemma, which extends Definition 39 to finite operation sequences.

LEMMA 16 Suppose, $H = I \bullet J$ is a legal history with $J = [o_1, \dots, o_n]$. Then, $I \equiv I \bullet J \bullet [o_n^{comp}, \dots, o_1^{comp}]$ and $I \bullet J \bullet [o_n^{comp}, \dots, o_1^{comp}]$ is legal.

Proof (Lemma 16): We proof the lemma by induction on the length n of J . If $n = 0$, the result is immediate since $H \equiv H \bullet [] \bullet []$ and $H \bullet [] \bullet []$ is legal by assumption.

Otherwise, let $I = K \bullet o_n$. By the induction hypothesis, $I \bullet K \bullet [o_{n-1}^{comp}, \dots, o_1^{comp}]$ is legal and $I \equiv I \bullet K \bullet [o_{n-1}^{comp}, \dots, o_1^{comp}]$. By Definition 39, it holds that $I \bullet K \bullet o_n \bullet o_n^{comp} \equiv I \bullet K$ and $I \bullet K \bullet o_n \bullet o_n^{comp}$ legal. Proposition 1 yields the lemma. $\square$

As introduced in Section 3.4.1.2, the compensation operation o^{comp} of an operation $o \in \Omega$ is computed using the *compensation function* $comp$ given in the specification of a cooperative activity type CAT . The requirement that $comp$ is total implies that there exists a unique compensation operation for every operation $o \in \Omega$.

DEFINITION 40 (COMPENSATION FUNCTION)

The *compensation function* $comp : \Omega \rightarrow \Omega$ is defined as follows: $comp$ is total and $\forall o \in \Omega : comp(o) = o^{comp} \Rightarrow o^{comp}$ is a compensation operation of o .

Definition 40 assumes that we can compute the compensation operation based on the original operation o . As a consequence, the compensation function cannot directly use information about the workspace state at the time operation o was executed. Only the input and output parameter values of o can be used in the specification of $comp$. This implies that the invocable operation types O of a CAT externalize as much information as needed for the computation of the compensation operations¹³.

The next example illustrates the concept of compensation developed so far. The example is based on the SEPIA scenario described in Section 3.4.2 and uses the compensation function for the SEPIA cooperative activity type described on page 88.

Example 42 (Conservative compensation approach) The left part of Figure 3.14 shows Alice's workspace state after she has imported operation q_1 and executed operations o_1 and o_2 ,

¹³In practice, this is no severe restriction since additional output parameters can simply be added to the operation types in O . As an advantage of this approach, no extra specification language constructs have to be introduced in order to specify the logging of additional information about the database state before or after the execution of an operation (like the BEFORE and AFTER statements in the VODAK modeling language [DIM95]).

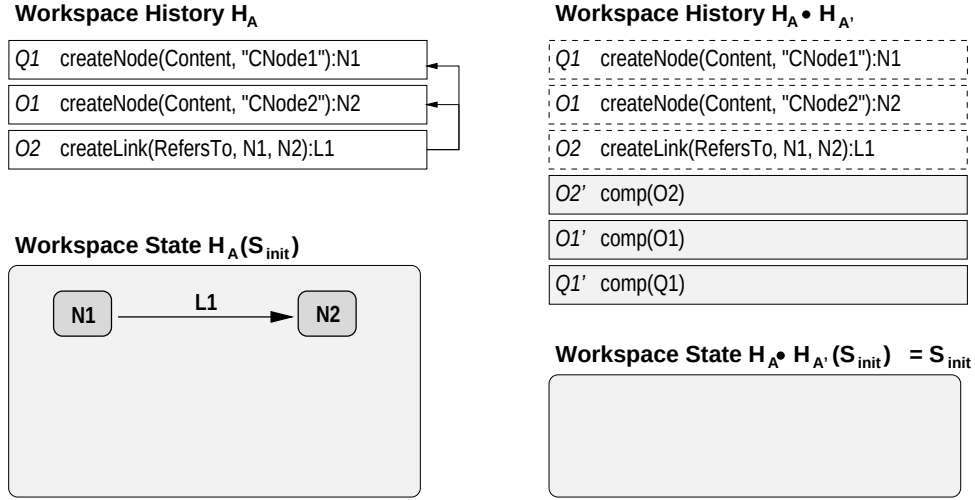


Figure 3.14: Conservative Compensation.

i.e., $H_A = [q_1, o_1, o_2]$. Assume, operation q_1 needs to be undone (either because of a merge conflict or because Alice wants to undo the imported operation q_1). The compensation function of the SEPIA *CAT* yields the following compensation operations:

$$\begin{aligned} q_1^{comp} &= \text{comp}(q_1) = \text{deleteNode}(N_1):[\text{Content}, \text{"CNode1"}] \\ o_1^{comp} &= \text{comp}(o_1) = \text{deleteNode}(N_2):[\text{Content}, \text{"CNode2"}] \\ o_2^{comp} &= \text{comp}(o_2) = \text{deleteLink}(L_1):[\text{RefersTo}, N_1, N_2] \end{aligned}$$

To compensate q_1 we have to compensate the operations in H_A in inverse order (Lemma 16), i.e., execute $H'_A = [\text{comp}(o_2), \text{comp}(o_1), \text{comp}(q_1)]$. The right part of Figure 3.14 shows the resulting state of the workspace and its history. $\square$

Looking more closely at this example, one can observe that in order to compensate an operation in a workspace history it is *not* necessary to compensate all successor operations in the history. For instance, operation o_1 need not be compensated in Example 42, since it is *independent* of q_1 , i.e., q_1 and o_2 commute backward. Therefore, we look at the question of when an operation can be compensated without compensating subsequent operations in the history.

DEFINITION 41 (DIRECTLY COMPENSATABLE OPERATION)

Suppose, $H = I \bullet o \bullet J$ is a legal history and $\text{comp}(o) = o^{comp}$. Operation o is *directly compensatable* in H iff $\forall p \in J : (o, p) \in bc$.

An operation o that is directly compensatable commutes with all successor operations in the history. This allows us to apply the compensation operation without first compensating the successor operations of o as the next lemma shows.

LEMMA 17 Let $H = I \bullet o \bullet J$ be a legal history, $\text{comp}(o) = o^{\text{comp}}$, and o be a directly compensatable operation in H . Then, the histories $H \bullet o^{\text{comp}}$ and $I \bullet J$ are legal and $H \bullet o^{\text{comp}} \equiv I \bullet J$.

Proof (Lemma 17): Using Proposition 2 (page 65), we get that $H \equiv I \bullet J \bullet o$ and history $I \bullet J \bullet o$ is legal. Since o^{comp} is the compensation operation of o , we obtain the lemma by Definition 39. $\square$

Example 43 (Directly compensatable operation) In Example 42, only operation o_2 is directly compensatable. Operations q_1 and o_1 are not directly compensatable since they do not commute backward with operation o_2 . If operation o_2 does not occur in the history (either because it has never been executed or is already compensated), both operations q_1 and o_1 become directly compensatable since q_1 commutes backward with o_1 . $\square$

Korth et al. [KLS90] introduced a different notion of compensatable operations. There, an operation o is directly compensatable if the compensation operation o^{comp} commutes backward with all successor operations of o , i.e., $\forall p \in I : (p, o^{\text{comp}}) \in bc$ instead of $\forall p \in J : (o, p) \in bc$ in Definition 39. This is a consequence of the definition of compensation operations in [KLS90] that uses state equality instead of state equivalence. For instance, in Example 42 Korth et al. would allow a *setBalance* operation to be a compensation operation for a *debit* operation.

We do not follow the approach of [KLS90], since (1) this requires to specify additionally backward commutativity between the invocable operations of a *CAT* and their compensation operations, (2) the specification of compensation functions becomes more complex since it becomes state-dependent, and (3) more operations need to be compensated than in our approach of compensation. For example, consider the history $[debit_1(a, 100); true, debit_2(a, 50); true]$ applied to a state with $balance(a) = 200$. Assume, the compensation operation for operation $debit_1$ is $setBalance(a, 200)$ (this is allowed in [KLS90]). Since $setBalance(a, 200)$ does not commute backward with $debit_2$, $debit_1$ is not directly compensatable (according to Definition 41), because $setBalance(a, 200)$ would also undo the effects of $debit_2$. In COACT, however, $setBalance$ is not allowed as a compensation operation for $debit_1$. We use $credit(a, 100)$ instead and, hence, $debit_1$ becomes directly compensatable.

We now define our approach to selective compensation as it is used in the COACT cooperative transaction model. Our general policy is to first identify a *minimal unit of compensation* within in a workspace history H . Starting with a set Q of operations to be compensated, we add recursively all subsequent operations of H to Q that are backward conflicting with operations in Q . Afterwards, we compute a *compensation sequence* that compensates exactly the operations in the unit of compensation.

DEFINITION 42 (UNIT OF COMPENSATION)

A subhistory I of history H is a *unit of compensation* iff $\forall o \in H, p \in I, p <_H o : (p, o) \notin bc \Rightarrow o \in I$. A subhistory I of H is a *unit of compensation for* $Q \subseteq op(H)$ iff I is a unit of compensation and $Q \subseteq op(I)$.

Definition 42 is analogous to the definition of closed subhistories (page 96) in the sense that the backward commutativity relation is used to compute dependent operations. The same relation is applied, but successor operations are considered instead of predecessor operations.

Like for closed subhistories, we can restrict the unit of compensation to those operations that are required to be compensated in order to maintain the legality of the resulting history. This leads us to the concept of a *minimal unit of compensation* which maximizes the number of operations preserved during compensation.

DEFINITION 43 (MINIMAL UNIT OF COMPENSATION)

The subhistory I of H is a *minimal unit of compensation* for $Q \subseteq op(H)$ iff it is a unit of compensation for Q and $\forall o \in I : (o \in Q) \vee (\exists p \in I : p <_I o \wedge (p, o) \notin bc)$.

The next lemma shows that there exists exactly one such minimal unit of compensation for a given set of operations. This allows us to uniquely identify in COACT the subhistory to be compensated, in case of merge conflicts or a user-initiated undo.

LEMMA 18 Let H be a history. For each set $Q \subseteq op(H)$ there exists exactly one subhistory I of H which is the minimal unit of compensation for Q .

Proof (Lemma 18): The proof is analogous to the proof of Lemma 13 (page 98). $\square$

After the minimal unit of compensation for a set of operations is determined, we compute the *compensation sequence* using the compensation function $comp$. The order of the compensation operations in the compensation sequence is inverse to the order of the operations in the minimal unit of compensation.

DEFINITION 44 (COMPENSATION SEQUENCE)

Let I be a unit of compensation for a set of operations $Q \subseteq op(H)$. Then, the *compensation sequence* C for the set Q is defined as $[o_n^{comp}, \dots, o_1^{comp}]$ with $o_i \in I$, $comp(o_i) = o_i^{comp}$, and $o_j^{comp} <_C o_i^{comp} \Leftrightarrow o_i <_I o_j$.

The next theorem shows that the compensation sequence C for a minimal unit of compensation I is legal and that the execution of C compensates only operations in I (and only those).

THEOREM 3

Let H be a legal history and I be (the minimal) unit of compensation for a set of operations $Q \subseteq op(H)$. Let C be the compensation sequence for Q . Let J be the restriction of H to operations in $op(H) \setminus op(I)$. Then, J and $H \bullet C$ are legal and $J \equiv H \bullet C$.

Proof (Theorem 3): Proposition 2 (page 65) yields that $H \equiv J \bullet I$ and $J \bullet I$ legal, since I is a minimal unit of compensation (cf. Definition 43). Thus, J is legal.

Suppose, $I = [o_1, \dots, o_n]$. Hence, the compensation sequence C is $[o_n^{comp}, \dots, o_1^{comp}]$ with $o_i^{comp} = comp(o_i)$. Lemma 16 yields that $J \equiv H \bullet C$ and $H \bullet C$ is legal. This proves the theorem. $\square$

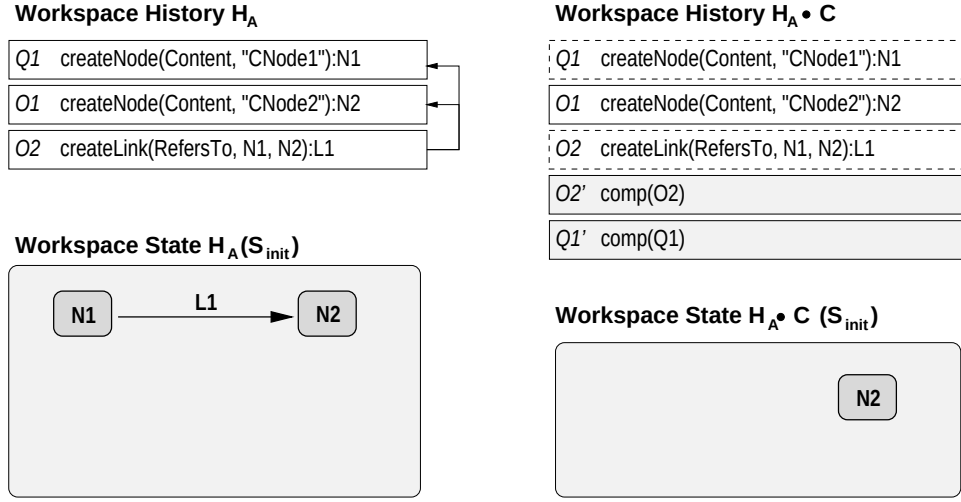


Figure 3.15: Selective Compensation.

To let users act autonomously within their workspaces, operations are compensated in CO-ACT only within a single workspace. If an operation occurring in more than one workspace history needs to be undone, the operation is *not* compensated in all workspaces in which it occurs. The operation is only compensated in the indicated workspace. The next example illustrates how selective compensation works in the COACT.

Example 44 (Selective compensation approach) Like in Example 42, Alice's workspace history is $H_A = [q_1, o_1, o_2]$ (cf. Figure 3.15). H_A contains the following units of compensation: $[], [o_2], [o_1, o_2], [q_1, o_1, o_2]$, and $[q_1, o_2]$.

Assume, Alice decides to compensate operation q_1 that created node N_1 , i.e., $Q = \{q_1\}$. The following subhistories of H_A are the only units of compensation for Q : $[q_1, o_1, o_2]$ and $[q_1, o_2]$. The latter is the minimal unit of compensation for Q . Thus, the compensation sequence C is $[o_2^{comp}, q_1^{comp}]$.

We then apply the compensation sequence to Alice's workspace history. The resulting history $H_A \bullet C = [q_1, o_1, o_2, o_2^{comp}, q_1^{comp}]$ is equivalent to the history $[o_1]$. The resulting workspace state after the compensation of Q is shown in the right part of Figure 3.15.

The operations q_1 and o_2 are only compensated in Alice's workspace, i.e., they are not compensated in other workspace histories in which they also occur. $\square$

In contrast to Example 42, operation o_1 need not be undone in our selective compensation approach in order to compensate q_1 . Only the minimal number of operations is compensated, i.e., those operations that transitively depend on those to be compensated.

3.5.4 CoACT Merge Algorithm

In practice, we need a constructive algorithm that implements the correct merging of histories in CoACT. Such a merge algorithm is presented in this section. The merge algorithm can be directly utilized to implement the CoACT exchange meta-operations, i.e., *import*, *delegate*, and *save*.

Afterwards, we present an extensive SEPIA example that illustrates in detail how the merge algorithm works and how cooperation is achieved in CoACT. In Section 3.5.4.3, we discuss several optimization possibilities and refinements of the basic merge algorithm.

3.5.4.1 Basic Merge Algorithm

The CoACT exchange meta-operations (cf. Definition 32 on page 93) take a set of operations selected in the source history and modify the target history in order to incorporate the effects of the selected operations into the destination workspace.

The general idea of the merge algorithm is therefore to *append* the selected operations S of the source history H_1 to the target history H_2 , and thereby preserving as many operations from the target history as possible. If relevant forward conflicts are detected, operations in the target workspace may be compensated if it is decided to discard the conflicting operations of the target history. Alternatively, only a subset of S can be incorporated into the target history. After the conflict resolution, the remaining operations from the source history are *re-executed* in the target workspace in order to construct the new target workspace state. The source workspace state and history are not affected by the merge.

In the following, we describe a basic algorithm that performs the correct merging of operations of a source history H_1 into a target history H_2 . This algorithm is a refinement of the algorithm presented in [WK96, WAN98].

BASIC MERGE ALGORITHM

1. Select some operations $S \subseteq op(H_1)$ from the source history H_1 .
-

2. Because S is not necessarily a consistent unit of work, complete S to a closed subhistory of H_1 . To ensure that only a minimal number of operations are exchanged, compute the *minimally closed subhistory* I of H_1 under S . We call I the *exchange history*.
-

3. Partition the exchange history I into a set $\mathcal{P} = \{I_1, \dots, I_n\}$ of closed subhistories I_k of I with $op(I_k) \cap op(I_j) = \emptyset$, $k \neq j$, and $op(I) = \bigcup_{k=1}^n op(I_k)$. We call the resulting histories I_k *exchange partitions*.

We treat each exchange partition as an *atomic unit of work*, i.e., either all or none operations of an I_k are incorporated into H_2 .

4. Next, check for *relevant merge conflicts* between the exchange partitions I_k and the target history H_2 , i.e., for each I_k , $k = 1, \dots, n$, compute

$$U_k := \{o \in (op(H_2) \setminus op(I)) \mid \exists p \in (op(I_k) \setminus op(H_2)) : (o, p) \notin fc\}.$$

- (a) If $U_k = \emptyset$, there exist no relevant forward conflicts between exchange partition I_k and the target history H_2 .
- (b) If $U_k \neq \emptyset$, a *decision* has to be made if the operations U_k should be undone in the target workspace in order to incorporate I_k ¹⁴.

If it is decided *not* to discard operations in U_k , then none of the operations in I_k can be merged into the target history H_2 (since we consider the merge of I_k with H_2 as atomic). This means that the merge for I_k is “aborted”: $U_k := \emptyset$ and $I_k := []$, i.e., $op(I_k) := \emptyset$.

5. After resolving the conflicts in the previous step, the *compensation sequence* C and the *re-execution sequence* E are computed.

- (a) To correctly compensate all operations $U := \bigcup_{k=1}^n U_k$ in H_2 , compute the *minimal unit of compensation* M in H_2 for U .
- (b) Compute the compensation sequence C in H_2 for the set U of operations to be compensated.
- (c) The set of operations to be merged into the target history is computed as follows: $\overline{P} := \bigcup_{k=1}^n op(I_k)$. Note that $\overline{P} \subseteq op(I)$, since merges for exchange partitions might have been “aborted” in Step 4.
- (d) Because $\overline{P}$ and H_2 can contain identical operations, i.e., $\overline{P} \cap op(H_2) \neq \emptyset$, eliminate all operations o from $\overline{P}$ with $o \in H_2$. Therefore, compute the restriction E of I to operations in $\overline{P} \setminus op(H_2)$. We call E the *re-execution sequence*.

¹⁴This decision can be taken either manually by a user or (semi-)automatically by an extra decision software module that implements certain pre-specified conflict resolution rules

6. The new state $S' = (H_2 \bullet C \bullet E)(S_{init})$ of the target workspace is constructed as follows.
 - (a) Execute the compensation sequence C in the target workspace.
 - (b) Execute the re-execution sequence E in the target workspace.
 - (c) Since compensated operations $o \in M$ and compensation operations $o^{comp} \in C$ together have no observable effect on the workspace state, we can ignore them in future merges. Therefore, compute the restriction H'_2 of H_2 to operations in $op(H_2) \setminus op(M)$. The new history of the target workspace is then $H'_2 \bullet E$.
-

In the following, we discuss some important characteristics of the basic merge algorithm described above. Afterwards, we prove that this algorithm works correctly, i.e., it produces a correctly merged history $H'_2 \bullet E$ with $H'_2 \bullet E \equiv H_2 \bullet C \bullet E$.

The rationale for the partitioning step (Step 2), which can be made optional, is as follows. If we allow only an atomic merge of the entire exchange history I into H_2 , i.e., to include all or none operations of I , the merge mechanism would be rather inflexible. In case of a merge conflict we either have to discard a large portion of H_2 or we cannot include any operation of I at all. Therefore, we allow for a *partial* merge, i.e., to merge only some exchange partitions I_k with H_2 .

Any exchange partition I_k contains at least one operation $s \in S$ selected in Step 1 because I is a minimally closed subhistory of H_1 under S . If $S = op(I)$, this is obvious. If $S \subset op(I)$, then each operation $o \in op(I) \setminus S$ precedes an operation $s \in S$ in history H_1 that is transitively backward conflicting with a o . Thus, o occurs in the same partition I_k as operation s . It follows, that the number of partitions is less or equal to the size of the set of selected operations S . The granularity of atomic merges and, hence, the granularity of the conflict resolution, is scalable by the number of partitions created. For example, an implementation of the merge algorithm can either compute the maximal number of exchange partitions or it can treat the exchange history I as a single exchange partition.

The exchange partitions I_k are pairwise independent closed subhistories of H_1 . Thus, each exchange partition I_k can be merged separately into H_2 . The conflict sets U_k , $k = 1 \dots n$, are accumulated in U before the algorithm computes the compensation sequence C and the re-execution sequence E . Therefore, the order of processing the I_k , $k = 1 \dots n$, in Step 4 does not matter for the result of the merge algorithm.

We assume that the workspace histories H_1 and H_2 do not change during the execution of the merge algorithm, as a result of a user executing operations in one of the workspaces or by concurrent merges. This is in line with almost all optimistic concurrency control protocols assuming that validation is performed serially.

Until Step 6, the merge process can be aborted since the merge algorithm does not change the source and target histories in Step 1-5. The state and the history of the target workspace

are changed for the first time when the compensation sequence and the re-execution sequence are applied in Step 6. Therefore, it is possible to extend the merge algorithm such that different merge alternatives can be considered in the conflict resolution before a merge is performed in the target workspace.

The basic merge algorithm can construct only a subset of all merged histories that are considered as correctly merged according to Definition 38¹⁵. Operations in the re-execution sequence E can never precede operations $o \in H_2$ before the merge, since we *append* E to the target history H_2 . In contrast to this, the definition of correctly merged histories allows to interleaf operations from both histories and to leave out some operations (recall that the empty history is always correctly merged). The merge algorithm described above tries to merge as many operations from S with the target history and minimizes the number of operations to be compensated in H_2 . To construct all correctly merged histories an extended merge algorithm would be required that can flexibly undo and later redo operations of the target history H_2 during the merge process.

Note that we do not need to consider the views of operations in the merge algorithm (like in the definition of correctly merged histories). This is not necessary, since the merge algorithm chooses an order for operations contained in both workspaces, based on the order of operations in the target workspace. For instance, consider Example 38 on page 104. When the workspace history $H_1 = [o_1, p_1]$ is incorporated into target history $H_2 = [p_1, o_1]$, the merge algorithm chooses the order $p_1 < o_1$ for the resulting merged history. Thus, for an implementation of history merging, we do not need to keep track of the views of operations.

In general, a higher degree of cooperation can be achieved if operations are exchanged more frequently between workspaces since then the workspace histories are synchronized more closely. This is reflected by the fact that the set $op(H_2) \setminus op(I)$ in U_k (Step 4) becomes smaller if $op(H_2) \cap op(I)$ becomes larger, i.e., the subhistory I of H_1 and H_2 overlap more. This results in less merge conflicts U_k . Thus, less operations in H_2 have to be discarded in order to re-execute E . Moreover, less operations have to be re-executed since the size of $\overline{P} \setminus op(H_2)$ becomes smaller if $op(H_2) \cap op(H_2)$ becomes larger. For instance, if $op(I) \subseteq op(H_2)$, no operations in H_2 have to be compensated and no operation of I has to be re-executed in the target workspace.

The next theorem shows that the basic merge algorithm is correct in the sense that it produces a correctly merged history H'_2 (cf. Definition 38). Since a correctly merged history is always legal if the input histories are legal (Theorem 2) the resulting history $H'_2 \bullet E$ is legal if H_1 and H_2 are legal.

THEOREM 4

If the workspace histories H_1 and H_2 are correctly merged histories, the merge algorithm computes a correctly merged history $H'_2 \bullet E$ with $H'_2 \equiv H \bullet C$.

Proof (Theorem 4): The idea of the proof is as follows: (1) We show that the restriction H'_1

¹⁵Having a criteria to describe the correctness of histories and an algorithm to construct some of these correct histories is similar to traditional transaction models. There, serializability is the correctness criteria and different concurrency protocols are employed that (only) allow subsets of serializable histories to be produced.

of H_1 to $\overline{P}$ (the operations that are actually incorporated into H_2) is a legal history. (2a) We prove that the minimal unit of compensation M in H_2 and $\overline{P}$ are disjoint. (2b) We show that the history $H'_2 \bullet E$, produced by the merge algorithm, contains exactly the operations $op(H'_1) \cup op(H'_2)$. (3) It remains to be shown that $H'_2 \bullet E$ is a correctly merged history, constructed from H'_1 and H'_2 . (4) Last, we prove that $H'_2 \equiv H \bullet C$. This yields the theorem.

(1) Let H'_1 denote the restriction of H_1 to operations $\overline{P}$. Since I is the minimally closed subhistory of H_1 under S and the I_k are closed and independent subhistories of I (Lemma 14), it follows that the subhistory H'_1 of H_1 is closed. By Theorem 1, H'_1 is legal because H_1 legal.

(2a) Obviously, $U \subseteq op(M)$. From the definition of U_k we get that that $U_k \cap P_k = \emptyset$. Thus, after step 4: $U \cap \overline{P} = \emptyset$ since $U = \bigcup_{k=1}^{k=n} U_k$ and $\overline{P} = \bigcup_{k=1}^{k=n} P_k$. Therefore, it is sufficient to show that $\overline{P} \cap (op(M) \setminus U) = \emptyset$. Assume, there exists a $p \in P_k : p \in op(M) \setminus U$. From Definition 43 follows that there exists an operation $o \in U$ with $(o, p) \notin bc$ and $o <_{H_2} p$. Since I_k is a closed subhistory of H_1 , this p must be contained in P_k and, hence, in $\overline{P}$. This is a contradiction to $U \cap \overline{P} = \emptyset$. Hence, $\overline{P} \cap op(M) = \emptyset$.

(2b) It is easy to see that $op(H'_2 \bullet E) = op(H'_2) \cup op(E) = op(H'_2) \cup op(H'_1)$, since $op(E) = \overline{P} \setminus op(H_2)$, $\overline{P} = op(H'_1)$, and $op(H'_2) \subseteq op(H_2)$. It remains to prove that the history $H'_2 \bullet E$ is a correctly merged history constructed out of H'_1 and H'_2 .

(3a) First, we prove that Condition 3 of Definition 38 is satisfied for the merged history $H'_2 \bullet E$, i.e., $\forall o \in (op(H'_2) \setminus op(H'_1)), p \in (op(H'_1) \setminus op(H'_2)) : (o, p) \in fc$. From the definition of U_k follows that $\forall o \in ((op(H_2) \setminus U_k) \setminus P), p \in (P_k \setminus (op(H_2) \setminus U_k)) : (o, p) \in fc$ holds after executing Step 4. Since $U = \bigcup_{k=1}^{k=n} U_k$ and $\overline{P} = \bigcup_{k=1}^{k=n} P_k$, it follows that $\forall o \in ((op(H_2) \setminus U) \setminus \overline{P}), p \in (\overline{P} \setminus (op(H_2) \setminus U)) : (o, p) \in fc$. Because $U \subseteq op(M)$, $op(H'_2) = op(H_2) \setminus op(M)$, $op(H'_1) = \overline{P}$, and $\overline{P} \cap op(M) = \emptyset$ (see (2a) above), it follows that $\forall o \in (op(H'_2) \setminus op(H'_1)), p \in (op(H'_1) \setminus op(H'_2)) : (o, p) \in fc$.

(3b) Next, we prove that Conditions 1 and 2 of Definition 38 are satisfied for the merged history $H'_2 \bullet E$. Both conditions are obviously satisfied for H'_2 since the order of operations in H'_2 is the same as in $H'_2 \bullet E$. Condition 1 is satisfied for H'_1 since $op(H'_1) \subseteq op(H'_2 \bullet E)$ (see (2b) above).

To prove Condition 2 for H'_1 , we need to show that $\forall o, p \in H'_1 : o <_{H'_1} p \wedge (o, p) \notin bc \wedge o \in view(p) \Rightarrow o <_{H'_2 \bullet E} p$. Let $H'_1 = K \bullet E$. Since E is a restriction of H'_1 , all operations in E appear in the same order as in H'_1 . Thus, it is sufficient to show that (a) $\forall o \in K, p \in E : o <_{H'_1} p \wedge (o, p) \notin bc \wedge o \in view(p) \Rightarrow o <_J p$ and (b) $\forall o, p \in K : o <_{H'_1} p \wedge (o, p) \notin bc \wedge o \in view(p) \Rightarrow o <_{H'_2} p$. (a) is certainly true since H'_1 is a closed subhistory of H_1 and $H'_1 = K \bullet E$. (b) If $o <_{H'_1} p \wedge o <_{H'_2} p$ this is immediate. If $o <_{H'_1} p \wedge p <_{H'_2} o$, then $(o, p) \in bc \vee o \notin view(p)$ has to hold. If $(o, p) \in bc$, we are done. If $(o, p) \notin bc$, then $o \notin view(p)$ since $o \in view(p)$ contradicts to the assumption that H_1 and H_2 are correctly merged history.

Thus, the history $H'_2 \bullet E$ constructed by the merge algorithm fulfills the conditions of correctly merged histories. By Theorem 2, $H'_2 \bullet E$ is legal since H'_1 and H'_2 are correctly merged histories.

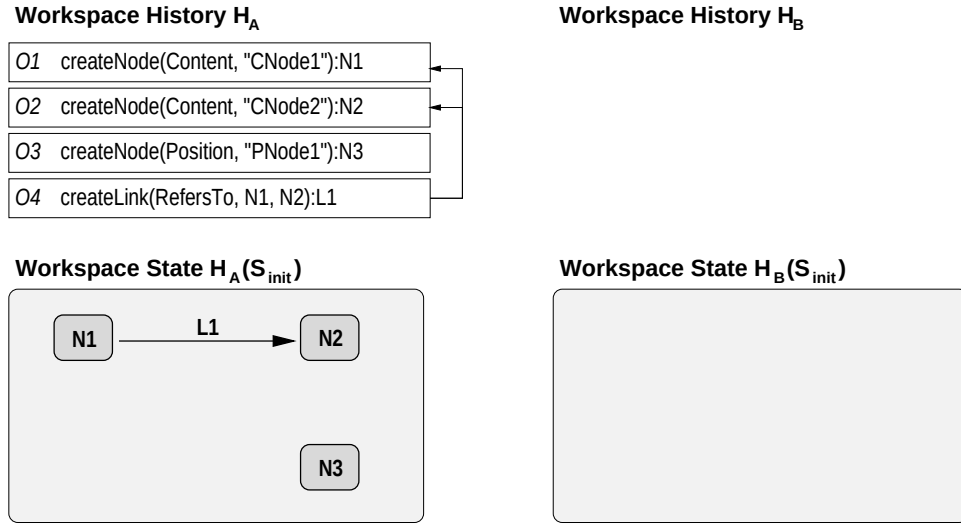


Figure 3.16: Situation before Delegation from Workspace A to Workspace B.

(4) It remains to show that $H'_2 \bullet E \equiv H_2 \bullet C \bullet E$. H'_2 is the restriction of H_2 to operations $op(H_2) \setminus op(M)$. Since M is the minimal unit of compensation for U , C is the compensation sequence for U , and H_2 is legal, it follows by Theorem 3 that $H_2 \bullet C$ is legal and $H'_2 \equiv H_2 \bullet C$. Using Proposition 1 it follows that $H'_2 \bullet E \equiv H_2 \bullet C \bullet E$ and $H_2 \bullet C \bullet E$ legal. $\square$

3.5.4.2 Example: COACT at Work

In this section, we illustrate how the merge algorithm works and how the COACT exchange meta-operations are used by co-workers within a cooperative activity. In the following, we assume two cooperating authors, Alice and Bob, acting in workspaces A and B within a cooperative activity of the SEPIA CAT (cf. Section 3.4.2). The cooperative authoring activity starts with an empty document, i.e., the initial state S_{init} of both private workspaces and the common workspace is the empty document, i.e., $S_{init} = S_{\perp}$.

Alice, acting in workspace A, performs the operations $[o_1, o_2, o_3, o_4]$ as shown in the left part of Figure 3.16. After the creation of content and position nodes N_1 , N_2 , and N_3 (operations o_1, o_2, o_3), the content nodes are connected by a reference link L_1 (operation o_4). The resulting states and histories of the workspaces A and B are depicted in Figure 3.16 (the common workspace is not shown).

Example 45 (Delegation from workspace A to workspace B) Alice decides to *delegate* the link L_1 to workspace B in order to stimulate contributions from her co-worker Bob. The set of selected operations S is $\{o_4\}$. In Step 2 of the merge algorithm, the minimally closed subhistory of H_A under S is computed resulting in the exchange history $I = [o_1, o_2, o_4]$. The exchange history I cannot be further partitioned, i.e., I represents the only exchange partition $I_1 \in \mathcal{P}$.

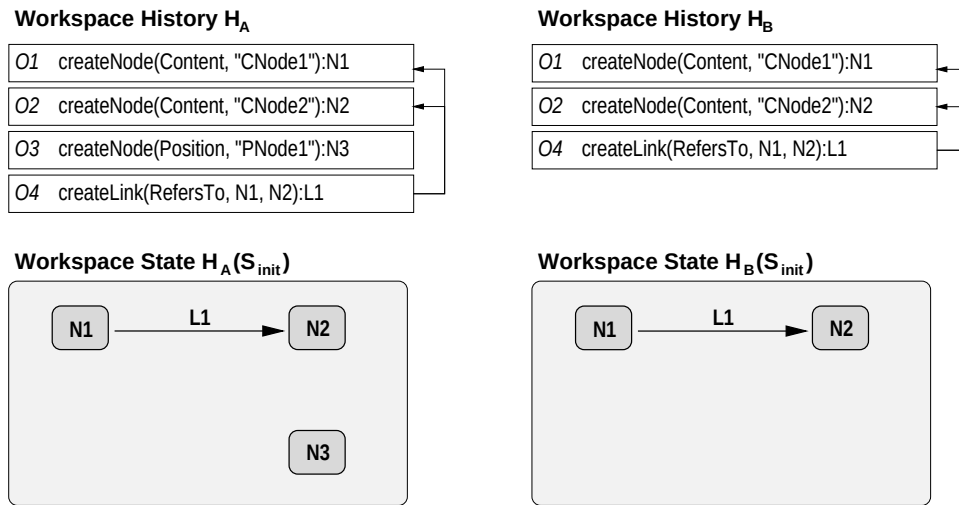


Figure 3.17: Situation after Delegation from Workspace A to Workspace B.

Afterwards, the exchange partition I_1 is sent to workspace B where Bob controls the integration of I_1 into his workspace B. First, the relevant forward conflicts between H_B and I_1 are computed. Due to the fact, that $H_B = []$, no conflicts occur, i.e., $U_1 = \emptyset$. Therefore, Bob can integrate I_1 into workspace B, i.e. $\overline{P} = \{o_1, o_2, o_4\}$. In Step 5 of the algorithm, the minimal unit of compensation is computed first. Since $U = U_1 = \emptyset$, the compensation sequence is $C = []$. Because H_B is empty, H_B and $\overline{P}$ do not contain identical operations. Thus, the re-execution sequence E is $[o_1, o_2, o_4]$.

To perform the actual integration, the operations in C and E are sequentially re-executed in workspace B. Afterwards, the workspace history H_B is set to $[o_1, o_2, o_4]$. The resulting histories of the private workspaces and their states are depicted in Figure 3.17. Note, the re-executed operations are now “replicated” among the workspaces A and B. $\square$

Let us assume both users have continued their work in isolation, resulting in the workspace histories presented in Figure 3.18. By means of operation o_5 Alice has added some content to the node N_1 , which can be a descriptive text. At the same time, Bob has put his own description to his “version” of node N_1 (operation p_1). Alice has deleted both link L_1 and node N_2 (operations o_6, o_7) because they did not fit with her plans. With operation o_8 , she changed the name of node N_1 to “Alice”. Meanwhile, Bob has created a new content node N_4 which he has linked to node N_2 (operations p_2 and p_3) and renamed node N_1 to “Bob” (operation p_4). Figure 3.18 shows the resulting workspace states and histories of A and B.

Both users have now changed the document content in conflicting ways, and we will explore the different merge alternatives in the next example.

Example 46 (Import from workspace B into workspace A) Alice plans to *import* the results of her co-worker and looks up (the history of) workspace B. Assume she wants to im-

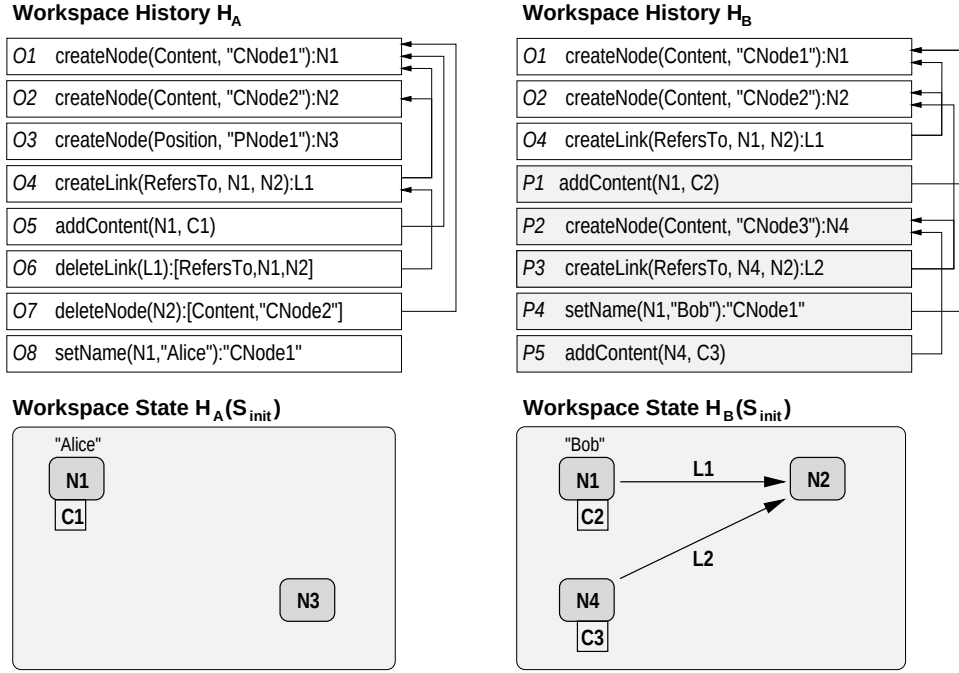


Figure 3.18: Situation before Import from Workspace B into Workspace A.

port all newly added links and nodes and name changes. Therefore, she (or the system) selects operations $S = \{p_2, p_3, p_4\}$. The merge algorithm computes the exchange history $I = [o_1, o_2, p_2, p_3, p_4]$. I contains two independent closed subhistories. Thus, I can be partitioned into exchange partitions $I_1 = [o_1, p_4]$ and $I_2 = [o_2, p_2, p_3]$, which can be processed separately in order to increase the flexibility during the merge process.

We start with I_1 . Operation o_1 is already present in H_A . Hence, no forward commutativity checking is required. Operation p_4 is processed next. It conflicts forward with o_8 . This yields $U_1 = \{o_8\}$. Now, Alice has to decide either to compensate o_8 or to abandon the integration of I_1 . In our example, she sticks to o_8 and the integration of the I_1 is “aborted”. Therefore, the system sets U_1 and P_1 to $\emptyset$.

The algorithm continues in Step 4 with exchange partition I_2 . Operation o_2 is already contained in workspace A and can be ignored. Operation p_2 commutes forward with the operations in H_A and can be merged with H_A without undoing operations in H_A . Operation p_3 (creation of the link between N_2 and N_4) conflicts forward with o_7 representing the deletion of N_2 . Thus, $U_2 = \{o_7\}$. Again, the controlling user (or decision module) has the choice between compensating o_7 (and all dependent operations in H_A) in order to incorporate I_2 or to abandon the integration of I_2 . We assume that the decision is to compensate o_7 .

The algorithm continues in Step 5 with $\bar{P} = \{o_2, p_2, p_3\}$ and $U = \{o_7\}$. The minimal unit of compensation for U in H_A is $M = [o_7]$. The compensation sequence $C = [comp(o_7)]$ with $comp(o_7) = createNode(Content, "CNode2"):N_2$. Afterwards, the re-execution sequence

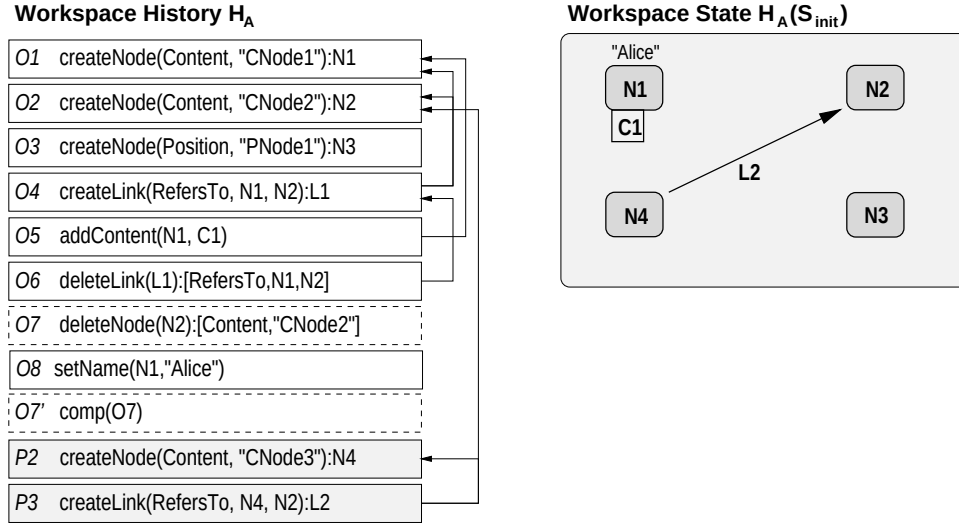


Figure 3.19: Situation after Import from Workspace B into Workspace A.

is computed. Since $H_A \cap \bar{P} = \{o_2\}$, this results in the re-execution sequence $E = [p_2, p_3]$. The resulting merged history is $H'_A = [o_1, o_2, o_3, o_4, o_5, o_6, o_8, p_2, p_3]$ which is equivalent to $H_A \bullet C \bullet E$.

In Step 6 of the algorithm, the merge is installed in workspace A. First, $comp(o_7)$ is executed in A. Then p_2 and p_3 are re-executed. The history of workspace A is set to H'_A . The resulting state of workspace A and its history are depicted in Figure 3.19. Workspace B is not affected by the merge. $\square$

The next example shows that correctly merged histories need not be serializable (cf. discussion on page 109 and Example 40).

Example 47 (Non-serializable correctly merged histories) Suppose, Alice creates a *Refers-To* link from node N_1 to node N_4 , and adds content to node N_4 . Figure 3.20 shows history H_A and the state of workspace A after Alice has executed operations o_9 and o_{10} .

The resulting workspace history H_A of Alice is not (conflict-)serializable with respect to “personal work units”, i.e., the operations generated by Alice and Bob. Let T_A denote the sequence of operations in H_A generated by Alice, i.e., $[o_1, o_2, o_3, o_4, o_5, o_6, o_8, o_9, o_{10}]$. T_B denotes the subhistory of H_B restricted to operations originally invoked by Bob, i.e., $[p_2, p_3]$. Looking at “transactions” T_A and T_B , we see that there exists a cycle $T_A \leftarrow T_B \leftarrow T_A$ in the dependency graph (cf. Section 3.5.2). The backward conflicts $o_2 \leftarrow p_3$, $p_2 \leftarrow o_9$, and $p_3 \leftarrow o_{10}$ that caused the cycle are shown as “thick” arrows in Figure 3.20. It follows that H_A is not serializable with respect to T_A and T_B . However, the history H_A is correctly merged and, hence, legal by Theorem 2. $\square$

This example shows that the merge algorithm is able to produce *non-serializable*, but still

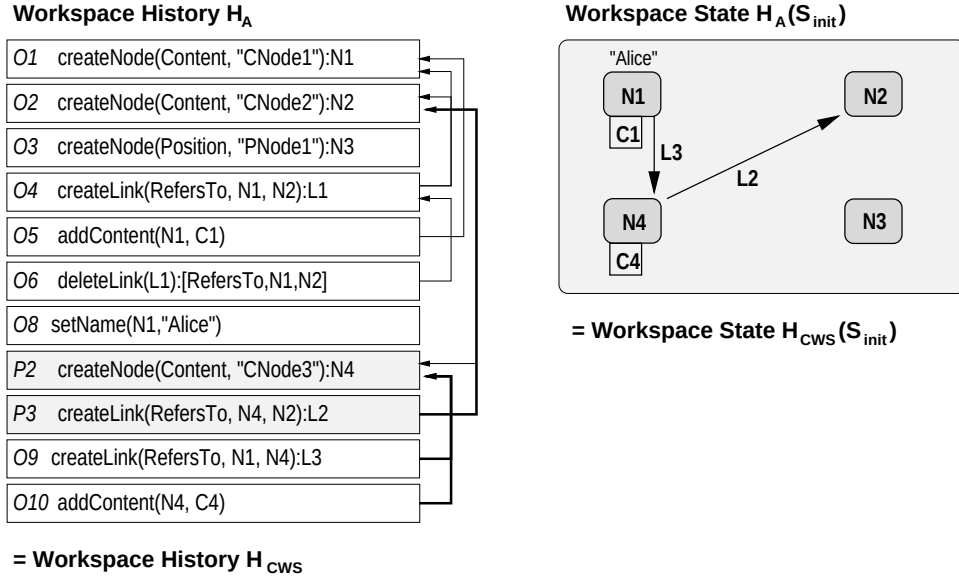


Figure 3.20: Situation before/after Save from Workspace A to the Common Workspace.

semantically correct, i.e., legal, histories. Thus, the isolation property is relaxed, but consistency of all workspaces is guaranteed.

The next examples illustrates the usage of the common workspace and of the *save* and *import* exchange meta-operations in the COACT model.

Example 48 (Saving to and importing from the common workspace) Alice wants to leave her office. To make her results publicly available, she *saves* her workspace (history) to the common workspace. Since the common workspace still is empty, i.e., $H_{CWS} = []$, the merge algorithm determines that H_A can simply be re-executed in the common workspace. Thus, $H_{CWS} = H_A$ after the *save*. The state of the common workspace is now the same as that of Alice's workspace as indicated in Figure 3.20.

Suppose, Bob is notified about Alice's *save* exchange meta-operation. He looks up the common workspace and decides to import all operations in H_{CWS} succeeding the last operation in H_{CWS} generated by him, i.e., $S = \{o_9, o_{10}\}$. The merge algorithm determines the exchange history $I = [p_2, o_9, o_{10}]$ as the only exchange partition. Since p_2 is already included in H_B and o_9 and o_{10} commute forward with all other operations in H_B not contained in H_{CWS} , $U = \emptyset$ and $\bar{P} = \{p_2, o_9, o_{10}\}$. The compensation sequence is $C = []$ and the re-execution sequence $E = [o_9, o_{10}]$. The state and history of workspace B after installing the merge are depicted in Figure 3.21.

As in the last example, the resulting merged history is non-serializable with respect to $T_A = [o_1, o_2, o_4, o_9, o_{10}]$ and $T_B = [p_1, p_2, p_3, p_4, p_5]$ as indicated by the “thick” arrows in Figure 3.21. $\square$

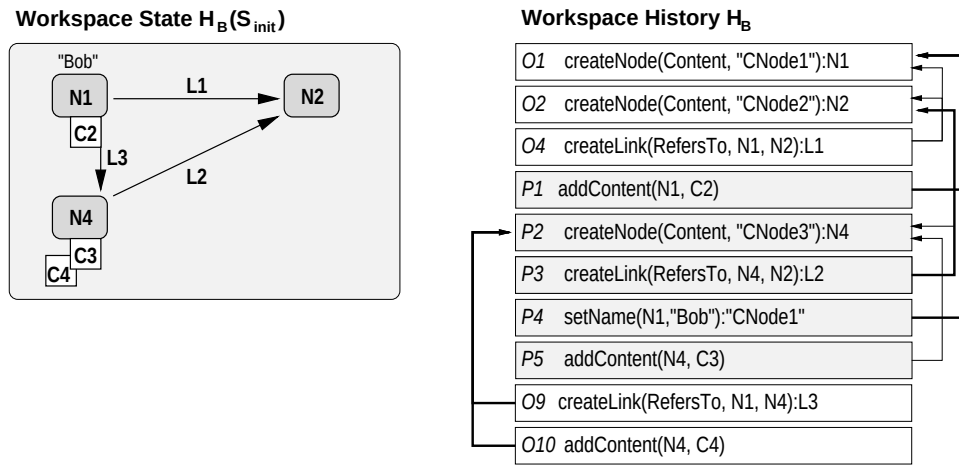


Figure 3.21: Situation after Import from the Common Workspace into Workspace B.

Other examples that illustrate the deployment of the COACT cooperative transaction model in the multi-user software development domain and for a cooperative calendar management application can be found in [WK96, WAN98] and [Kar98], respectively.

3.5.4.3 Optimization and Refinements of the Merge Algorithm

The merge algorithm described on page 117 provides a basis for the implementation of a number of variants of merge algorithms and exchange meta-operations. For example, it is possible to adapt the basic algorithm to dedicated environments, exploiting their special characteristics. In this section, we discuss some optimizations and refinements of the basic merge algorithm.

3.5.4.3.1 Modified Merge Algorithm. The basic merge algorithm does not make strong assumptions on access to the histories during its execution. The algorithm can be decomposed into two independent parts. The first part, consisting of steps 1 and 2, can be executed solely with access to the source history. The second part of the algorithm, consisting of steps 4, 5, and 6, accesses only the target history. The exchange history is the glue between the two parts of the algorithm. Thus, the basic algorithm allows a distributed implementation, where workspaces and workspace histories are stored in a decentralized way. For mobile environments, this might be the only way to implement the merge module (see Section 4).

In a more centralized environment (like the TRANSCOOP system, cf. Section 5.1), the workspaces and their histories can be stored in a central database. Since a merge module then has permanent access to both the source and the target history, there is a potential to optimize the basic algorithm. In the following, we outline a modified merge algorithm that can yield better performance. The basic idea of the optimization is to eliminate as many operations as early as possible during the merge process. We assume that the restrictions of histories can be computed in an efficient way.

MODIFIED MERGE ALGORITHM

1. Select S from the source history H_1 .

Then, remove all operations from S that are already contained in the target history H_2 , i.e., $S' = S \setminus op(H_2)$.

2. Calculate H'_1 as the restriction of H_1 to operations in $op(H_1) \setminus op(H_2)$.

Then, compute the minimally closed subhistory I' of H'_1 under S' .

3. Compute the exchange partitions I'_k of I' (analogous to the basic algorithm).
-

4. Calculate H''_2 as the restriction of H_2 to operations in $op(H_2) \setminus op(H_1)$.

For each I'_k , compute $U'_k := \{o \in H''_2 \mid \exists p \in I'_k : (o, p) \notin fc\}$.

The rest of Step 4 is analogous to the basic algorithm.

5. Compute the minimal unit of compensation M' in H''_2 for $U' := \bigcup_{k=1}^{k=n} U'_k$ and the compensation sequence C' in H''_2 for U' .

The re-execution sequence E' is the restriction of I' to operations in $\overline{P'} := \bigcup_{k=1}^{k=n} op(I'_k)$.

6. Execute the compensation sequence C' and the re-execution sequence E' of the target workspace.

The history of the target workspace is set to $H'_2 \bullet E'$, where H'_2 is the restriction of H''_2 to operations $op(H_2) \setminus op(M')$.

Like the basic merge algorithm, the modified merge algorithm computes a correctly merged history $H'_2 \bullet E' \equiv H_2 \bullet C' \bullet E'$ but makes use of the following observation. Since some operations in S may be already present in H_2 , they need not be considered for the merge. Therefore, in the first step, we eliminate all operations from S that are contained in H_2 . Based on the same argument, all other operations in H_1 that are already contained in H_2 need not be exchanged. Also, operations in H_2 already contained in H_1 need not be compensated nor re-executed. The

modified algorithm therefore operates on the modified histories H'_1 and H'_2 which contain less operations than H_1 and H_2 , in case the histories H_1 and H_2 are not disjoint.

In the following, we briefly compare both merge algorithms. Since $S' \subseteq S$, $op(H'_1) \subseteq op(H_1)$, the computation of the closed subhistory I' becomes more efficient. The exchange history I' usually contains fewer operations than I , i.e., $P' \subseteq P$. Therefore, partitioning is less expensive. In Step 4, H''_2 and P'_k are disjoint. Since $op(I') \subseteq op(I)$, less forward conflict checks have to be executed. Generally, the resulting set U' is smaller than U since $op(I') \subseteq op(I)$. Because $op(H''_2) \subseteq op(H_2)$ and $U' \subseteq U$, the computation of M' and the compensation sequence C' become more efficient. The elimination of duplicates in Step 5 can be omitted since $\overline{P'} \cap op(H_2) = \emptyset$, whereas in the original algorithm $\overline{P}$ and $op(H_2)$ need not be disjoint.

It is easy to see that the performance gain offered by the modified algorithm is greater, the larger the overlap of H_1 and H_2 is. For example, if $S \subseteq op(H_2)$, the modified algorithm determines already in Step 1 that nothing has to be done. The original algorithm recognizes this in Step 5 during the elimination of duplicate operations from $\overline{P}$.

To further optimize Step 5 of the modified algorithm, the re-execution sequence E' can be directly set to $I_{k_1} \bullet \dots \bullet I_{k_n}$ where $k_1, \dots, k_n$ is a permutation of $1, \dots, n$. Thus, we do not need to compute the restriction of I' to $\overline{P'}$. This changes the order of the operations re-executed (compared to the order in H_1) but does not affect the correctness of the algorithm since Proposition 2 yields that $E' \equiv I_{k_1} \bullet \dots \bullet I_{k_n}$.

3.5.4.3.2 Modified Partitioning Step. The (optional) partitioning step in both merge algorithms can be further refined to yield a larger number of atomic exchange partitions. This enables a finer granularity of conflict resolution.

Example 49 (Sub-partitioning) Consider the workspace history H_A as shown in Figure 3.22. Assume, the set of selected operations is $S = \{o_4, o_5\}$. In the basic merge algorithm, the minimally closed subhistory of H_A under S is $I = [o_1, o_2, o_4, o_5]$. Step 3 yields $I_1 = I$ as the only exchange partition. If only one of o_4 and o_5 has a relevant forward conflicting operation in the target history, it is not possible to merge only the non-conflicting operations of S (and their relevant predecessors in I) into the target history. The reason is that we consider the merge of an exchange partition as atomic, i.e., either all operations of the exchange partition are included in the target history or none of them.

However, I_1 can be further decomposed into *non-disjoint* sub-partitions $I_1^1 = [o_1, o_2, o_4]$ and $I_1^2 = [o_1, o_5]$ as depicted in Figure 3.22. Sub-partitions can then be merged independently into the target history. For example, if o_5 has a relevant forward conflict, it is still possible to include o_4 into the target history. Note that each sub-partition includes at least one operation $s \in S$. $\square$

In the following, we outline how the partitioning step can be refined by adding sub-partitioning. If an exchange partition I_k contains more than one operation of S , i.e., $|op(I_k) \cap S| > 1$,

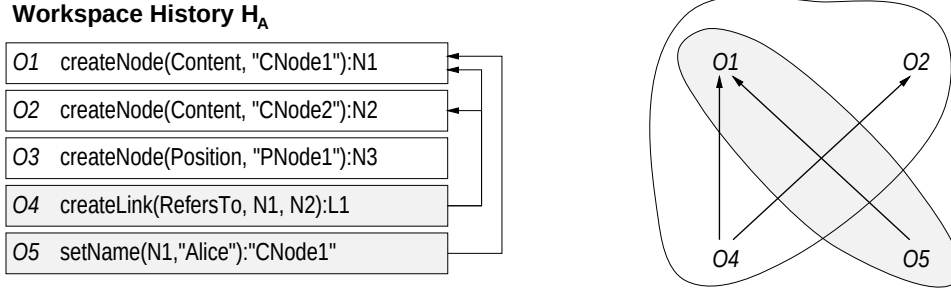


Figure 3.22: Sub-Partitioning of Exchange Partitions.

then I_k is a candidate for sub-partitioning. If there exist two operations $s, t \in (op(I_k) \cap S)$ such that $s <_{I_k} t$, but s is not in the minimally closed subhistory of I_k under $\{t\}$, then I_k can be sub-partitioned. A naive approach for sub-partitioning I_k is to compute for all operations $o_l \in (op(I_k) \cap S)$ the minimally closed subhistories I_k^l of I_k under $\{o_l\}$. These minimally closed subhistories form the set $\mathcal{Q}_{I_k}$ of all possible sub-partitions of I_k . If there exist $I_k^l, I_k^m \in \mathcal{Q}_{I_k}$ with $op(I_k^l) \subseteq op(I_k^m)$, then I_k^l is removed from $\mathcal{Q}_{I_k}$.

The remaining sub-partitions are then merged with the target history. However, care has to be taken since sub-partitions are no longer disjoint. This is no problem in the described merge algorithms, since, in Step 5, the re-execution sequence E is computed using the exchange history I . Thus, no operation can appear twice in E .

How an even finer merge granularity can be achieved is described in [KTW96b, KTWK98]. There, we allow merges of arbitrary prefixes of exchange partitions I_k with the target history. This may result in situations in which some prefix I_k' of I_k is incorporated into the target history, but none of the selected operations $s \in (op(I_k) \cap S)$. Because the user explicitly requested the merge of operations $s \in S$ with the target history, it appears to be meaningless to include operations of I_k into the target history if the operations $s \in (op(I_k) \cap S)$ are not included, too. Therefore, in this thesis we treat exchange (sub-)partition as the *smallest atomic unit of work* in CoACT.

3.5.4.3.3 Optimization of Compensation and Re-execution. To optimize the compensation and re-execution process, we can utilize the concepts of *transparent* operations [MRKN92, LMWF94, WK96]. Transparency is the essential property of “read” accesses, reflecting the fact that a workspace state is not altered in an *essential* way. We say that an operation is transparent if for any history H where o is legal the state directly before o is equivalent to the state resulting from the execution of o . Thus, a transparent operation does not affect the behavior of subsequent operations in a history. A special case of transparent operations are *read-only* operations.

DEFINITION 45 (TRANSPARENT AND READ-ONLY OPERATIONS)

An operation o is *transparent* iff $\forall$ histories $H : H \bullet o \text{ legal} \Rightarrow H \equiv H \bullet o$. An operation o is *read-only* iff $\forall$ histories $H : H \bullet o \text{ legal} \Rightarrow H(S_{\perp}) = (H \bullet o)(S_{\perp})$.

LEMMA 19 Suppose, operation o is read-only. Then, o is transparent.

Proof (Lemma 19): The lemma follows immediately from Lemma 2 (page 58). $\square$

Notice that this definition applies to operations rather than to operation types (or operation invocations). Operation types (or operation invocations) may only sometimes modify the workspace state in an observable way. For example, a *debit* from a bank account can modify the balance of the account but a return value of *false* indicates that the account was not debited. Definition 45 can be easily extended to operation types (or operation invocations) if we require that all possible instantiations of operation types (or operation invocations) are transparent or read-only, respectively.

The next lemmas show the relationship of transparent operations and commutativity.

LEMMA 20 Let $H \bullet o$ and $H \bullet p$ be legal histories and o and p transparent operations. Then, $(o, p) \in bc$ and $(o, p) \in fc$.

Proof (Lemma 20): Since o and p are transparent it follows that $H \bullet o \equiv H \bullet p \equiv H$. It follows that $H \bullet o \bullet p$ and $H \bullet p \bullet o$ are also legal (Proposition 1 on page 59). Since $H \bullet o \equiv H \bullet p \equiv H$ it follows that $H \bullet o \bullet p \equiv H \bullet p \bullet o \equiv H$. Hence $(o, p) \in bc$ and $(o, p) \in fc$. $\square$

LEMMA 21 Suppose, $H \bullet o$ and $H \bullet p$ are legal histories and o is a transparent operation. Then, $(o, p) \in lfc$.

Proof (Lemma 20): The proof follows directly from the fact that $H \bullet o \equiv H$ and histories $H \bullet o$ and $H \bullet p$ are legal. Thus, $H \bullet o \bullet p$ legal and $H \bullet o \bullet p \equiv H \bullet p \bullet o$. This yields $(o, p) \in lfc$. $\square$

These two lemmas can be helpful for the specification of the commutativity tables of a cooperative activity type. Another important observation about transparency is that the compensating operation for transparent operations is the *noop* operation. The *noop* operation is defined to be legal in any state, and its before-state is always equivalent to its after-state.

DEFINITION 46 (NOOP OPERATIONS)

The *noop* operation is defined as follows: $\forall$ legal histories $H : H \bullet noop \text{ legal} \wedge H \bullet noop \equiv H$.

LEMMA 22 Let o be a transparent operation. Then $o^{comp} = noop$.

Proof (Lemma 22): Assume $H \bullet o$ is a legal history. Then $H \bullet o \bullet noop$ is legal since *noop* is legal in any state. Because $H \bullet o \equiv H$, it holds that $H \bullet o \bullet noop \equiv H \bullet noop$. Since $H \bullet noop \equiv H$, we get that $H \bullet o \bullet noop \equiv H$. $\square$

From Lemma 22 follows that, in Step 6 of the merge algorithm, transparent operations contained in the minimal unit of compensation M never have to be compensated. Thus, their compensation operations need not to be included in the compensation sequence C .

Furthermore, it is possible to ignore transparent operations already while computing the minimal unit of compensation, as the next example shows.

Example 50 (Transparent operations in selective compensation) Suppose, $H = [o_1, o_2, o_3]$ is a legal history, with $o_1 = \text{debit}(a, 100):\text{true}$, $o_2 = \text{balance}(a):200$, $o_3 = \text{debit}(a, 100):\text{true}$. The minimal unit of compensation of $\{o_1\}$ in H is $[o_1, o_2, o_3]$, since $(o_1, o_2) \notin bc$ and $(o_2, o_3) \notin bc$. However, in the restriction $H' = [o_1, o_3]$ of H to non-transparent operations, the minimal unit of compensation for o_1 is $[o_1]$, since $(o_1, o_3) \in bc$.

In the former case, the transparent operation o_2 induces a dependency among o_1 and o_3 , whereas o_1 and o_3 are independent in H' . Thus, by ignoring transparent operations, the compensation sequence $[o_3^{\text{comp}}, o_2^{\text{comp}}, o_1^{\text{comp}}]$ could be reduced to $[o_1^{\text{comp}}]$. $\square$

The ignorance of transparent operations $\bar{T}$ in the target history H_2 already during the computation of minimal units of compensation $\bar{M}$ reduces the size of the compensation sequence C . This results in compensating fewer operations in the target history, in case merge conflicts occur. Note that, in order to guarantee the correctness of the merge algorithm, the new target history needs then to be set in Step 6(c) to $\bar{H}_2' \bullet E$, where $\bar{H}_2'$ is the restriction of H_2 to operations in $op(H_2) \setminus (\bar{M} \cup \bar{T})$ (instead of $op(H_2) \setminus M$).

Transparency can also be used to optimize the re-execution process in the target workspace in certain cases. Since transparent operations do not change the workspace state in an essential way, they need not be re-executed in Step 6 of the merge algorithm to construct the new state of the target workspace. This approach is motivated by the following lemma.

LEMMA 23 Let H be a legal history and H' be the restriction of H to operations that are not transparent. Then H' is legal and $H \equiv H'$.

Proof (Lemma 23): The lemma follows directly from the successive application of Definition 45 and Proposition 1. $\square$

Observe that operations in a history record their output parameter values. Thus, we need not *compute* the output parameter values of transparent operations during the re-execution process. Hence, in Step 6(b) of the merge algorithms, the re-execution process of $E = [o_1, \dots, o_n]$ can be re(defined) as follows. For $i = 1, \dots, n$: If $o_i(\vec{in}_i):\vec{out}_i$ is a transparent operation simply take $\vec{out}_i$ as the output parameter values, without executing o_i . Otherwise, re-execute o_i , i.e., invoke $o_i(\vec{in}_i)$, to reflect the state changes of o_i in the target workspace and return the result. Note that in Step 6(c) the new target history still is set to $H_2' \bullet E$, i.e., *all* operations in E are included in the target workspace history.

In the following, we outline optimization possibilities applicable for cooperative applications that are only interested in the consistency of workspace states after a merge, but not in the return values of re-executed operations.

One optimization is to ignore all transparent operations in the re-execution phase. Thus, we can first compute the restriction E' of E to non-transparent operations and then re-execute the shortened re-execution sequence E' instead of E .

A more radical approach that might be viable for certain classes of cooperative applications is to ignore transparent operations in general during the merge process. This means that transparent operations are not logged in the histories of the workspaces. Thus, they are not taken into account during a merge process. Obviously, transparent operations then do not need to be considered in the specification of the merging relations M and the compensation function $comp$ of a cooperative activity type CAT .

Further optimizations of the compensation and re-execution process are possible. For instance, we can utilize the concept of *masking* operations [WK96, Rak97, LMWF94]. Informally, a (non-transparent) operation p masks an operation o if, whenever $H \bullet o \bullet p$ is legal, $H \bullet o \bullet p \equiv H \bullet p$ and $H \bullet p$ is legal, too. Therefore, operations in the target history that are masked by operations in the re-execution sequence E need not necessarily be compensated. Moreover, operations in the re-execution sequence E masked by subsequent operations in E need not necessarily be re-executed.

Example 51 (Masked operations) Consider a legal target history $H_2 = I \bullet o \bullet J$ with $o = setName(N_1, \text{“Alice”})$. Suppose, the compensation sequence C is $[o^{comp}]$ and the re-execution sequence E is $[p, q]$, where $p = setName(N_1, \text{“Bob”})$ and $q = setName(N_1, \text{“Charlie”})$. Since q masks both p and o , it is not necessary to execute o^{comp} and p —only q needs to be re-executed to construct the new state of the target workspace. The resulting history $H_2 \bullet q$ is equivalent to $H_2 \bullet o^{comp} \bullet E$ and $I \bullet J \bullet E$. $\square$

However, care must be taken in this optimization, since in a history $I \bullet o \bullet J \bullet p$ the operation o may influence the behavior of operations in J , i.e., despite o is masked by p . Only if all operations in J commute (right) backward with operation o , we can be sure that $I \bullet o \bullet J \bullet p \equiv I \bullet J \bullet p$.

3.5.4.3.4 Compaction of Histories. If a cooperative activity is long-lived, the workspace histories can grow rather large. It is easy to see that the merge algorithm performs better if the workspace histories can be shortened.

Since the basic COACT model allows each co-worker to maintain his own private “view” on the cooperative activity CA in his private workspace throughout the lifetime of CA , it is difficult to shorten the workspace histories without affecting the correctness of history merging. This is different to optimistic concurrency control where, at some point in time, intention lists for committed transactions can be discarded [HW88]. There, a committed transaction can be forgotten when it is sufficiently old by appending its intention list to the common prefix which is represented compactly as a materialized version.

One possible approach to shorten histories is to log only non-transparent operations in the workspace histories (cf. the discussion in Section 3.5.4.3.3 on page 132).

Another approach is to introduce special *synchronization points* in the CoACT model. The basic idea is that synchronization points in workspace histories mark equivalent prefixes of the histories. These equivalent prefixes can be ignored during history merging, since they identify equivalent states.

The usage of synchronization points seems most appropriate if there is a coordination protocol implemented that regularly synchronizes the workspace histories using our merge algorithms and *pre-defined rules* for conflict resolution. Examples of cooperative applications that use such coordination protocols are distributed real-time groupware systems that regularly multicast modifications from one site to the other sites (cf. Section 2.3.8.1).

Example 52 (Synchronization points) Consider the following two histories of workspaces A and B including a synchronization point $sync_1$: $H_A = I \bullet sync_1 \bullet J$ and $H_B = I' \bullet sync_1 \bullet K$. The synchronization point $sync_1$ in the histories identifies that $I \equiv I'$. If some operations from J are merged into H_B , we only have to consider the subhistories J and K in the merge process, since we know that the states of workspaces A and B were equivalent at the synchronization point. $\square$

The introduction of synchronization points has the potential to significantly improve the performance of the merge algorithms. On the other hand, additional coordination protocols (like the coordination model CoCoA, cf. Section 3.6.1) are required to enforce the synchronization of two or all workspace histories in a cooperative activity. Additionally, it must be ensured that no operation before a synchronization point can be compensated (e.g., due to user-initiated selective undo), since then the synchronization point becomes invalid.

If every workspace history in a cooperative activity contains the same synchronization points and, hence, equivalent prefixes, it is possible to discard the prefixes of the workspace histories identified by the synchronization point. Like in [HW88], the prefix is then represented as a permanent version of the workspace states.

3.5.5 Asymmetric Relations in the Context of CoACT

In this section, we present a generalized definition of correctly merged histories that uses the asymmetric commutativity relations rbc and lfc .

3.5.5.1 Generalization of History Merging

In Section 3.5.2, we have described how two workspace histories can be merged correctly, using the symmetric bc and fc relations (cf. Definition 38 on page 106).

In the following, we present a generalized condition for correctly merged histories that makes use of the asymmetric commutativity relations rbc and lfc . At the same time, we extend the definition of correctly merged histories such that an arbitrary number of histories can be merged at once.

DEFINITION 47 (GENERALIZED CORRECTLY MERGED HISTORY)

A legal history $H = [o_1, \dots, o_n]$, where $o_i \in \text{view}(o_{i+1}), i = 1, \dots, n \Leftrightarrow 1$, is a *generalized correctly merged history*.

Let $H_i, i = 1, \dots, n$, be mergeable and generalized correctly merged histories. A merged history M , consisting of operations $op(M) \subseteq \bigcup_{i=1}^n op(H_i)$, is considered as *generalized correctly merged* if the following conditions are satisfied:

1. $\forall p \in M : (\exists \text{ history } H_i : o, p \in H_i \wedge o <_{H_i} p \wedge (o, p) \notin rbc \wedge o \in \text{view}(p)) \Rightarrow o \in M \wedge o <_M p$
2. $\forall o, p \in M : (o <_M p \wedge \nexists \text{ history } H_i : o, p \in H_i) \Rightarrow (o, p) \in lfc$
3. $\forall o, p \in M : (\exists \text{ history } H_i : o, p \in H_i \wedge o <_{H_i} p \wedge o \notin \text{view}(p) \wedge p \notin \text{view}(o) \wedge (p, o) \notin lfc) \Rightarrow o <_M p$

The first condition is a generalization of Conditions 1 and 2 in Definition 38. It states that if an operation p appears in the merged history, all other preceding right backward conflicting operations in its source history, that are in the view of p , must precede p in the merged history (cf. Proposition 4 on page 71).

The second condition is a generalization of Condition 3 in Definition 38. It states that two operations contained in the merged history but not originating from the same source history must commute left forward, in the order they appear in the merged history.

The third condition states that the order of operations that are in a deferred-update relationship have to be preserved in the merged history if both operations do not commute left forward in the opposite order. This condition does not occur in Definition 38, since it is not needed if we consider the symmetric forward commutativity relation fc , instead of lfc . Conditions 2 and 3 together ensure that Proposition 6 (page 78) is applicable.

THEOREM 5

A generalized correctly merged history satisfying Definition 47 is legal. All generalized correctly merged histories consisting of the same operations are equivalent.

Proof (Theorem 5): The proof of the theorem is analogous to the proof of Theorem 2. Assume the correctly merged history M was constructed from the legal histories $H_i, i = 1, \dots, n$. Let $O_\cap = (op(M) \cap \bigcap_{i=1}^n op(H_i))$. Let K denote the restriction of M to operations in $O_\cap$, I_i denote the restriction of H_i to operations in $op(M) \setminus O_\cap$, and J_i denote the restriction of H_i to operations in $op(H_i) \setminus op(M)$. It follows that $op(M) = O_\cap \cup \bigcup_{i=1}^n op(I_i)$, since operations in J_i are not contained in the merged history M .

(1) We show that the reordering $L_i = K \bullet I_i \bullet J_i$ of H_i is legal, $H_i \equiv L_i$, and $K \bullet I_i$ legal. It is easy to see that $op(L_i) = op(H_i)$. For all operations that appear in a different order in H_i and L_i , i.e., $p <_{H_i} o \wedge o <_{L_i} p$, it is ensured by Condition 1 in Definition 47 that $(o, p) \in rbc \vee o \notin \text{view}(p)$. Condition 3 ensures that in case o and p are in a deferred-update

relationship $(o, p) \in lfc$ holds ($(p, o) \in lfc$ holds since H_i is legal). Thus, by Proposition 4 it follows that L_i is legal and $L_i \equiv H_i$. Since $K \bullet I_i$ is a prefix of L_i , $K \bullet I_i$ is legal, too.

(2) Let G be a history with $op(G) = P \setminus O_\cap = \bigcup op(I_i)$ and $<_G \supseteq (< \cup \bigcup_{i=1}^n <_{I_i})$, where $< = \{(o, p) \mid \exists o \in I_i, p \in I_j, i \neq j : (o, p) \in lfc \wedge (p, o) \notin lfc\}$. Since Condition 2 in Definition 47 ensures $(o, p) \in lfc$ for all operation pairs (o, p) with $o \in H_i, p \in H_j, i \neq j, o <_M p$, Proposition 6 yields that all histories $M' = K \bullet G$ are legal and equivalent.

(3) The last step is to show that the correctly merged history M (and any other correctly merged history with the same operations as M) is legal and equivalent to $M' = K \bullet G$. It is easy to see that M' consists of the same operations as M . Conditions 1 and 3 in Definition 47 ensure that M is legal and $M \equiv M'$ (analogous to above). The same holds for any correctly merged history with the same operations as M . $\square$

Theorem 5 holds for any binary relations $uip \subseteq rbc \subseteq \Omega \times \Omega$ and $du \subseteq lfc \subseteq \Omega \times \Omega$. A special case are the symmetric relations $bc \subseteq rbc$ and $fc \subseteq lfc$. If more restrictive relations than rbc or lfc are used, more intra-workspace dependencies and more inter-workspace conflicts are observed. This restricts the merged histories that are considered as correctly merged but does not affect the legality of the resulting merged histories.

Using Definition 47 and Theorem 5, several merge algorithms can be constructed, based on the merge algorithms and optimization described in Section 3.5.4, that exploit the asymmetric commutativity relations and allow more correctly merged histories than those using symmetric commutativity relations. To that extent, it is useful to re(de)fine our notion of consistent units of work as follows:

- A subhistory I of a history H is closed iff $\forall o \in H, p \in I : o <_H p \wedge o \in view(p) \wedge (o, p) \notin rbc \Rightarrow o \in I$.

This is the refinement of Definition 33 (page 96) for the asymmetric rbc relation.

- A subhistory I of a legal history H is closed under $Q \subseteq op(H)$ iff I is closed and $Q \subseteq op(I)$ (cf. Definition 33).
- A closed subhistory I of H is minimally closed under $Q \subseteq op(H)$ iff $\forall o \in I : (o \in Q) \vee (\exists p \in I : o <_I p \wedge o \in view(p) \wedge (o, p) \notin rbc)$

This is the refinement of Definition 34 (page 98) for the asymmetric rbc relation.

It turns out that a closed subhistory I (as defined above) of a legal history H is legal (cf. Theorem 1, page 96). The proof is analogous to the proof of Theorem 1, using Proposition 4 instead of Proposition 2.

For the resolution of conflicts, our notion of compensation can be re(de)defined as follows:

- A subhistory I of history H is a unit of compensation iff $\forall o \in H, p \in I, p <_H o : p \in view(o) \wedge (p, o) \notin rbc \Rightarrow o \in I$.

This is the refinement of Definition 42 (page 114) for the asymmetric rbc relation.

- A subhistory I of H is a unit of compensation for $Q \subseteq op(H)$ iff I is a unit of compensation and $Q \subseteq op(I)$ (cf. Definition 42).
- A subhistory I of H is a minimal unit of compensation for $Q \subseteq op(H)$ iff it is a unit of compensation for Q and $\forall o \in I : (o \in Q) \vee (\exists p \in I : p <_I o \wedge p \in view(o) \wedge (p, o) \notin rbc)$.

This is the refinement of Definition 43 (page 115) for the asymmetric rbc relation.

- The compensation sequence is defined as in Definition 44 (page 115).

It turns out that Theorem 3 (page 115) holds with the re(de)defined definitions above. The proof is analogous to that of Theorem 3, but uses Proposition 4 instead of Proposition 2.

3.5.5.2 Other Relations than Commutativity in the Context of CoAct

In [KTW96b, KTWK98], we examined the usage of weaker semantic relationships than commutativity for history merging, namely *recoverability* (rcv)¹⁶ [BR92] and *ignores* (ign)¹⁷ [KTW96b]. In contrast to commutativity relations, rcv and ign do not require that the workspace states resulting from different execution orders of a pair of operations are equivalent.

The *recoverability* relation considers whether an operation is independent of a previous operation, i.e., the previous operation is not needed to guarantee a legal execution.

DEFINITION 48 (RECOVERABILITY RELATION)

$\forall o, p \in \Omega : (o, p) \in rcv$ iff $\forall$ histories $H_1, H_2 : H_1 \bullet H_2$ is legal $\wedge H_1 \bullet o \bullet H_2 \bullet p$ is legal $\Rightarrow H_1 \bullet H_2 \bullet p$ is legal.

The *ignores* relation describes whether an operation becomes illegal if another operation is executed previously.

DEFINITION 49 (IGNORES RELATION)

$\forall o, p \in \Omega : (o, p) \in ign$ iff $\forall$ histories $H_1, H_2 : H_1 \bullet o \bullet H_2$ is legal $\wedge H_1 \bullet H_2 \bullet p$ is legal $\Rightarrow H_1 \bullet o \bullet H_2 \bullet p$ is legal.

In particular, we investigated the usage of recoverability in [KTW96b, KTWK98] to identify consistent units of work within a single workspace history, and the use of the ign relations to reduce the number of relevant conflicts and, hence, the amount of operations that have to be discarded during a merge. Since rcv considers the removal of operations from histories and ign their insertion, the rcv relation may be used to compute closed subhistories and units

¹⁶The recoverability relation is introduced in [BR92] to decrease the delay involved in processing non-commuting operations of different transactions, while still avoiding cascading aborts.

¹⁷The ignores relation is the complement of the invalidated-by relation introduced in [Her86, Her90]. Invalidated-by is a serial dependency relation [Her86, Her90] that can be used in the context of optimistic concurrency control to define legal histories and serializability of ACID transactions.

of compensation (replace *bc* or *rbc*) and the *ign* relation may be used to determine conflicts between operations generated in different workspaces (replace *fc* or *lfc*).

Unfortunately, it turned out from our analysis that both relations suffer from severe anomalies that may occur during history merging [KTW96b, KTWK98]. For this reason, we think that the usage of (right) backward commutativity and (left) forward commutativity is the best choice for our model. These relations limit the number of conflicts (e.g., they introduce less conflicts than state-independent commutativity), while on the other hand preserving the correctness of history merging.

3.6 Discussion

In this chapter, we have presented a formal description of the COACT model and its history merging mechanism. The COACT cooperative transaction model provides a flexible framework to support cooperation in interactive, multi-user environments.

As mentioned in the introduction, COACT focusses on transactional aspects of cooperative activities. It is only concerned with the correctness of sharing and exchange of persistent data among cooperating users. Organizational aspects of cooperative activities were addressed in the TRANSCOOP project by the COCoA coordination model (cf. Section 1.3). COCoA allows to specify application-dependent constraints that describe the organizational structure and control flow within cooperative activities.

In the following, we briefly describe the building blocks of COCoA, and outline how the COCoA coordination model is integrated with the COACT cooperative transaction model. Afterwards, we assess the COACT model with respect to the requirements of cooperative applications described in Section 2.2 and compare it to the advanced transaction models presented in Section 2.3.

3.6.1 Organizing the Work Process: The COCoA Coordination Model

The COCoA coordination model [FEdB96, FEdba97, FEdB98], developed at the University of Twente, offers a rich set of primitives to describe the organizational structure of a cooperative activity. COCoA stands for *Coordinated Cooperation Agendas*.

COCoA is designed on top of the formal specification language LOTOS/TM [EFdB96, EF94, dBEV95], which in turn is based on the ISO standard LOTOS [BB89, vEVD89] and the TM database specification language and design tools, developed at the University of Twente [BBdB⁺96, vKSA⁺95, FvKS94, BdBZ93]. The main concepts of COCoA are described in the following.

3.6.1.1 Specifying Organizational Aspects of Cooperative Activities

Broadly speaking, the organizational aspects in a COCoA specification determine who is allowed to do what at which state of the cooperative activity execution.

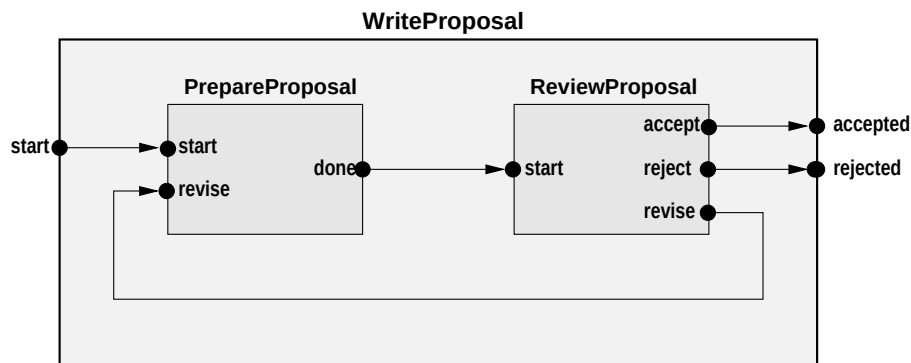


Figure 3.23: Graphical Representation of the Sample CoCoA Scenario `WriteProposal`.

The key primitive in the CoCoA coordination model is a *step*. Intuitively, a step represents an organizational unit of work within a cooperative activity. It describes which actors can be active in a step and defines which actions can be performed by these actors.

A CoCoA scenario may consist of a single step, but more commonly, multiple steps are involved. Steps can be combined in various ways to form the organizational structure of the work process. The constructs to group steps range from sequential and repetitive execution of steps to nested and parallel steps. Note that CoCoA's concept of steps is much broader than that found in traditional workflow applications. A single step can deal with more than one user, and each user can be involved in more than one step at the same time.

The set of actions that are controlled by a step include data operations, CoACT's exchange meta-operations, as well as so-called *communications*. Communications are used to initiate step transitions, to terminate steps, and to dynamically instantiate parameterized sub-steps. Any action has to be explicitly *enabled* before a user that is active in this step can execute it. If two steps are active at the same time, this leads to an accumulation of the enabled actions for the actors involved. The step concept and the enabling mechanism of CoCoA are complemented by the possibility to specify *user roles*.

In the following, we introduce a simple cooperative authoring example (`WriteProposal`) to illustrate the concepts of CoCoA introduced above. It illustrates a simple sequence of two steps (`PrepareProposal` and `ReviewProposal`), which can be activated several times. The graphical representation of the example is given in Figure 3.23; Figure 3.24 sketches the corresponding CoCoA scenario specification¹⁸.

First, the possible user roles in the CoCoA scenario are specified. In the example in Figure 3.24, four user roles are declared: `actor`, `author`, `reviewer`, and `editor`. Note that user roles can be hierarchically structured. For example, an `editor` is also an `author`.

¹⁸Please note that, for the sake of clarity, the sample specification in Figure 3.24 includes only those statements that are necessary to understand the main concepts of CoCoA, described in this section. Moreover, the syntax has been slightly changed. An executable CoCoA specification for a SEPIA cooperative authoring activity can be found in [WEF⁺96]. Other extensive CoCoA examples are given in [FEdB96, FEdba97, FEdB98].

```

1  SCENARIO WriteProposal INCLUDES SepiaSchema
2
3  USER ROLES
4      actor,
5      author  ISA actor,
6      editor  ISA author,
7      reviewer ISA actor
8
9  COMMUNICATIONS
10     ProposalFinished(), ProposalAccepted(),
11     ProposalRejected(), ReviseProposal()
12
13  PROCEDURE(reviewer, editor, author)[IN start OUT accepted, rejected]
14
15     STEP PrepareProposal[IN start, revise OUT done]
16         ON start, revise ENABLE
17             author:
18                 // enabled data operations:
19                 createNode, removeNode, createLink, ...
20                 // enabled exchange meta-operations:
21                 IMPORT FROM author, DELEGATE TO author
22                 IMPORT FROM CWS,    SAVE TO CWS
23                 // enabled communications:
24                 WHEN editor ISSUES ProposalFinished() DO done
25             ENDON
26         END//Step
27
28     STEP ReviewProposal[IN start OUT accept, reject]
29         ON start ENABLE
30             reviewer: ...
31             WHEN reviewer ISSUES ProposalAccepted() DO accept
32             WHEN reviewer ISSUES ProposalRejected() DO reject
33             WHEN reviewer ISSUES ReviseProposal()   DO revise
34         ENDON
35     END//Step
36
37     // transition rules:
38     ON start                DO PrepareProposal.start
39     ON PrepareProposal.done  DO ReviewProposal.start
40     ON ReviewProposal.accept DO accepted
41     ON ReviewProposal.reject DO rejected
42     ON ReviewProposal.revise DO PrepareProposal.revise
43
44     END//Procedure
45
46  HISTORY RULES
47
48     CONFLICT
49         createNode(ntype,name):node
50         createLink(ltype,fnode,tnode):node
51     BACKWARD node <> fnode AND node <> tnode
52     FORWARD  true
53
54     ...
55
56  END//WriteProposal

```

Figure 3.24: Sketch of a Sample CoCoA Specification for Cooperative Authoring.

Afterwards, the possible communications are declared in the CoCoA specification.

The *procedure definition* lists a number of steps and a number of *transition rules*. The latter describe how the former are chained together, and define the coarse-grained control flow within a cooperative activity. Each step definition defines, among others, its entry and exit interaction points at which interactions with other steps can take place. For example, the step `PrepareProposal` has two entry points (`start` and `end`) and a single exit point (`done`); these are declared in square brackets in Figure 3.24. Entry interactions activate the step; exit interactions deactivate it. The procedure definition itself has an entry point and at least one exit point. In our example, the CoCoA scenario can terminate with two possible outcomes: `accepted` and `rejected`.

The allowed interactions among steps are defined in the transition rules (line 38-42 in Figure 3.24). The transition rules govern which state transitions are allowed in the CoCoA scenario. In particular, they specify how the interaction points of the steps relate to each other (see also Figure 3.23).

At the beginning of a cooperative activity, all steps are inactive. When the cooperative activity is initiated, the entry point of the procedure definition is enabled. In our example, this is the `start` entry point. The transition rule in line 38 of the example specifies that the entry point `start` of the step `PrepareProposal` is triggered in this case. This results in the activation of step `PrepareProposal`.

A step specification defines which data operations, exchange meta-operations, and communications are enabled for the user roles in an active step. Data operations and exchange meta-operations are enabled inside a step for a specific user role. The enabled operations can be invoked by the respective actors any number of times, and in any given order. The enabling takes place only when the required interaction takes place, as defined in the step. When a step terminates all permissions issued are withdrawn. Only by invoking (enabled) communications, one or more other steps can become activated or deactivated.

Consider, the `PrepareProposal` step in Figure 3.24. If the `start` or the `revise` entry point is triggered, users of role `author` are allowed (1) to invoke the listed data operations and (2) to execute the following exchange meta-operations: *delegate* or *import* work to/from co-authors and *save* or *import* work to/from the common workspace.

Only a user that fills the `editor` role is allowed to execute the `ProposalFinished()` communication. If this communication is issued, it results in the deactivation of the step `PrepareProposal`, since it triggers the exit interaction point of this step (line 24 in Figure 3.24). The transition rule in line 39 is then triggered, which results in the activation of step `ReviewProposal`.

Note that in most transactional workflow models, step-like structures serve also to define transactional boundaries. This is not the case in CoCoA, where steps only help to organize the work in smaller units. In other words, the deactivation of a step does not imply any “commits” or “aborts” of data operations invoked while the step was active.

3.6.1.2 Constraints on Single Workspaces

Steps are used to describe the division of the work within a cooperative activity, but CoCoA provides additional means to detail what is and what is not allowed in a cooperative activity.

With regard to single workspaces the application designer can specify a set of *execution order rules* [FEdB96, FEdba97, FEdB98]. These rules pose workflow-like restrictions on the order and existence of operations in a workspace history. Execution order rules can also be used to define *termination constraints* to further constrain the enabling of communications.

In CoCoA, a language-based specification mechanism for the execution order rules is used. The application designer specifies certain grammars, by means of extended regular expressions on operation invocation pattern. The allowed sequences of operations within workspaces are equal to the words in the generated language. To specify this language multiple grammars can be used. Their combination results in the desired restrictions on sequences of operations that are allowed in the workspaces. In contrast to [Ska89, NRZ92], our mechanism avoids dead ends caused by interdependencies of different grammars [KTW⁺96d, FEdB96].

Execution order rules have to be enforced for each private workspace and the common workspace separately [RKT⁺95a, KTW⁺96d, FEdB96]. This allows the enforcement of execution order rules even for those applications, where the workspaces are not permanently connected, e.g., in mobile environments [KTW97b, KTW97a]. For more details on execution order rules, see [FEdB96, FEdba97, FEdB98, KTW⁺96d].

3.6.1.3 Integration of the CoACT Cooperative Transaction Model with CoCoA

Within the TRANSCOOP project, we have integrated the CoACT model into the CoCoA specification language.

CoCoA offers language constructs to specify the input for the CoACT cooperative transaction model, i.e., a cooperative activity type. A CoCoA specification refers to a database schema (*SepiaSchema* in our example). The invocable operation types of a CoACT cooperative activity type, which are a subset of the operation types available in the database schema, are declared by the **DATA OPERATIONS** section in a CoCoA specification (not shown in Figure 3.24). The merging relations, i.e., backward and forward commutativity among the invocable operations, are specified in the **HISTORY RULES** section in a CoCoA specification. Figure 3.24 shows the specification of the (complement of the) backward and forward commutativity relations between *createNode* and *createLink* operations of the sample SEPIA cooperative activity type (cf. Section 3.4.2).

Additionally, the invocation of data operations and CoACT's exchange meta-operations in a cooperative activity can be constrained using CoCoA's step mechanism and user roles. Data operations and exchange meta-operations have to be explicitly enabled in the steps before they can be invoked by a user. CoCoA also provides a mechanism for the specification of queries over workspace histories to select data operations for CoACT's exchange meta-operations.

Moreover, CoCoA allows to specify dynamic consistency constraints on data operation

invocations by means of the execution order rules. Termination constraints can be used to link the state of execution order rules to the step transitions in a CoCoA procedure definition.

This integration of transactional aspects with organizational aspects within a single specification language enables the specification of application-dependent execution constraints at different system layers, e.g., organizational constraints concerning the coordination of work and transactional constraints on the workspace level. It makes CoCoA suitable for specifying free forms of cooperation, as found in cooperative authoring, while still being able to specify more restrictive forms of cooperation found in workflow applications [FEdbA97].

For a comprehensive overview of the CoCoA coordination model, its concepts, and its specification language, we refer the reader to [FEdbB96, FEdbA97, FEdbB98].

3.6.2 Requirements Revisited

In the following, we discuss how COACT fulfills the requirements for a cooperative transaction model as described in Section 2.2.

Support for private and shared data (requirement D) is addressed in COACT by introducing a common workspace and a private workspace for each participant in a cooperative activity. Each private workspace contains private “versions” of data objects and constitutes a consistent configuration of the data. Workspaces can be realized by a dynamic replication scheme. How our model can support mobile users is described in Chapter 4.2.

Support for the exchange of (tentative) results (requirement C) is realized by the history merging mechanism. By exploiting the backward commutativity relation, we make sure that only consistent units of work are exchanged among workspaces. By exploiting the forward commutativity relation, we ensure that no inconsistencies are introduced, when operations are re-executed in another workspace. The concept of identical operations enables us to establish a close cooperation among co-workers because certain conflicts in the cooperation can be ignored. We have shown that correctly merged histories remain legal, i.e., workspaces are always consistent, although we allow non-serializable histories. Cooperation primitives are realized by means of the *import*, *export*, and *save* exchange meta-operations in COACT.

Requirement A (relaxed atomicity) is addressed in our model by providing durability of workspaces. Durability of workspaces is an orthogonal issue to the other COACT concepts and is not required by history merging. Recovery of failures can easily be addressed in our model with traditional database recovery concepts. Before an operation is executed in a workspace, the corresponding history entry is written to durable storage (write-ahead-logging) (cf. Section 5). If a failure occurs in the private workspace, its most recent state can be reconstructed using the workspace history (forward recovery). This process can be optimized by periodically checkpointing the modified objects. Note that a failure within a single private workspace never affects other workspaces.

To support requirement B (retraction of decisions), we have introduced a flexible undo mechanism in COACT. By exploiting the intra-workspace dependencies between operations

induced by backward conflicts in a workspace history, it is possible to selectively undo, i.e., compensate, operations. The system computes the minimal compensation sequence for the selected operations in the history and then executes the compensation sequence to re-construct the workspace state.

Coordination of individual and joint work (requirement E) is not directly supported by COACT, since in the TRANSCOOP project we have decided to separate transactional and organizational aspects. COACT focusses on the transactional aspects, whereas the COCOA coordination model addresses the organizational aspects within cooperative activities.

Other requirements like support for mutual awareness of co-workers and additional communication facilities like e-mail or audio/video connections required for direct negotiation of co-workers are not covered by COACT, since it aims at synchronizing the cooperative access to shared persistent data. Nevertheless, the COACT model provides an application-independent nucleus to realize a wide spectrum of cooperative applications.

In COACT, we achieve cooperation by means of exchanging operations among workspaces (instead of data objects). One may argue that the exchange of results in terms of data or documents, respectively, is more natural from a user's viewpoint. However, the exchange of data objects can be mapped to the exchange of a respective set of operations contained in a workspace history, e.g., using appropriate graphical user interfaces or the history querying facilities of COCOA. On the other side, the usage of operations and the exploitation of their semantics enable a higher degree of cooperation than only exchanging data objects (and, hence, going back to a read/write model).

Due to the fact that the COACT model allows to identify consistent units of work dynamically at runtime, it is well-suited for cooperative environments in which the occurrence of conflicts cannot be determined at specification time, e.g., highly interactive and creative design processes like cooperative authoring.

3.6.3 Related Transaction Models

The COACT model shares some ideas with the advanced transaction models described in Section 2.3. Figure 3.25 illustrates these relationships. In the following, we discuss the relationships of the COACT cooperative transaction model to other advanced transaction models.

Like in semantic concurrency control [Kor83, GM83, SS84, Wei88a, Wei89a, RC97], we exploit the semantics of operations defined on data objects. We use the backward commutativity relation to determine intra-workspace dependencies. Unlike conventional transaction models, this allows us to dynamically determine operation sequences, that can be considered as consistent units of work, i.e., “transactions”, and that can be exchanged independently among workspaces. In this respect, our approach is similar to the one found in the operating system Gutenberg [VRS86] where action dependencies (which result from interprocess communication between kernel objects and shared use of capability directories) are used to define recoverable computations.

The idea of using views of operations is similar to causal precedence in Lamport clocks

[Lam78]. There a “happens before” relation is employed that is used to define dependent and concurrent operations. In [Lam78], two events that are in a “happens before” relation are always considered as dependent (this is similar to the assumption in real-time cooperative editing systems [SE98b]). The same definition of causality is used in the virtual time approach and its implementation, the Time Warp mechanism [Jef85]. The Time Warp mechanism has been first proposed for controlling sequences of events in distributed simulation and was later extended to other application domains such as concurrency control in DBMS [JM86, GRL94].

In the COACT cooperative transaction model, an operation p is considered as dependent on o only if o is in the view of p and p does not commute (right) backward with o . In [LAA93], a similar approach that exploits the message semantics is used to reduce the amount of rollback in Time Warp. Interestingly, Time Warp's notion of global virtual time [Jef85] seems to be applicable to COACT. The global virtual time corresponds to a moving commitment horizon and may be used to define synchronization points in COACT to compact workspace histories (cf. Section 3.5.4.3.4).

To check whether the exchange of operation sequences among different workspaces maintains consistency of workspaces, we apply the forward commutativity relation. In this respect, our approach is similar to semantics-based optimistic concurrency control [Her86, Her90], with the difference that we work on multiple copies of data objects in different workspaces. In contrast to conventional transaction models, we do not assume that different histories (of transactions) are disjoint. Thus, more histories are legal in COACT than in [Her86, Her90], which enforces isolation and atomicity properties of transactions. We have illustrated in this chapter that the more two histories overlap, the better the merge process works.

To resolve conflicts that might occur during history merging, we apply the concept of compensation. This is similar to SAGAs [GMS87, GMGK⁺91, CR92] and the open-nested transaction model [WS92, MRKN92, MRW⁺93, WDSS93, MR93, Wei91] where (partial) rollback is implemented by executing compensating subtransactions for all subtransactions executed after the one to be compensated. Our approach to compensation is more flexible. We use the backward commutativity relation to determine minimal units of compensation and compensation sequences. This ensures that only operations are compensated that are dependent on the one to be compensated, thus minimizing loss of work.

Realtime group editors (like GROVE and REDUCE [EG89, SYZC97]) use a technique called operational transformation to resolve conflicts among independent conflicting operations [EG89, SYZC97, SE98b]). The basic idea is to include both conflicting operations in the history instead of only one of them. This is possible if an (application-dependent) transformation function can be found that transforms the effects of the successor operation in such a way that its effects are integrated in the workspace without interfering with the predecessor operations. For cooperative text editing such functions are known [SE98b]. For example, consider the insertion of a character into a string $str = \text{“There will be student here!”}$. If $o = \text{insert}(str, 14, \text{“a ”})$ and $p = \text{insert}(str, 21, \text{“s”})$ and both operations are in a deferred-update relationship, they conflict forward since they modify the same string. However, it is possible to integrate both op-

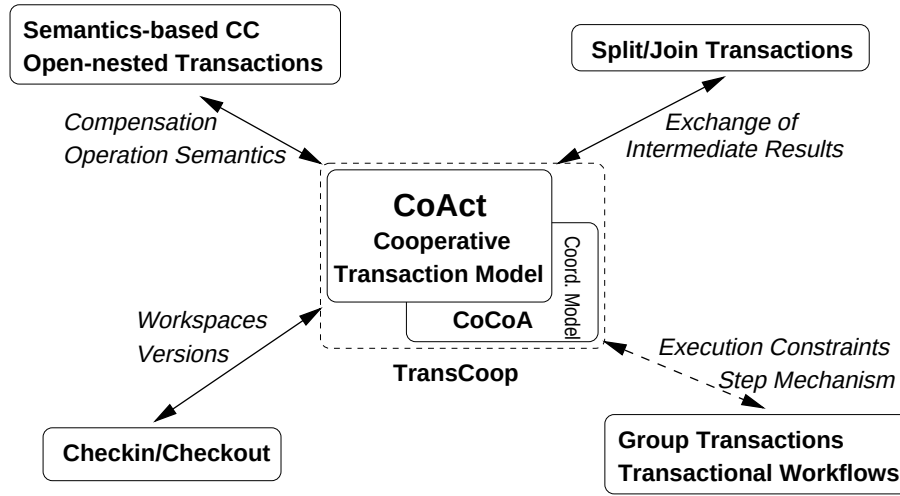


Figure 3.25: Relationship of the CoACT Model to Selected Advanced Transaction Models.

erations into a workspace, e.g., in the order $o < p'$, where p' denotes the transformed operation $p' = \text{insert}(\text{str}, 23, \text{"s"})$. The resulting string is “There will be a students here!”, which is considered as syntactically correct in [SYZC97, SE98b] but is obviously not semantically correct. In our opinion, it is better to flag a conflict that has to be explicitly resolved than to introduce inconsistencies at the semantic level. Moreover, such operation transformation functions exist only for simple cooperative text editing systems.

With respect to the workspace concept our approach is close to the check-out/check-in models [Roc75, KW84, Tic85, Kat90, KBGW91, Kai95, KF87, Hon88, KPS89]. In the basic check-out/check-in model, checked-out objects are reserved for exclusive access until a later check-in. This may result in blocking of objects over long periods of time. In COACT, we avoid this behavior by providing each user with his own (implicit) consistent configuration in his private workspace. Approaches which combine the check-out mechanism with versioning go into a similar direction but they suffer from the fact that their support for the required merging of different versions is only limited. Most merge mechanisms provided by versioning schemes force the user to choose one of the generated versions and drop the others. Otherwise, the user has to perform the cumbersome task of integrating the different versions manually. In contrast to this, COACT is able to merge the work of different users automatically by using the provided operation semantics. Only in cases, where users have done irreconcilable work, interference is necessary. Even in this case, our merge algorithm can propose the user different alternatives, which minimize the amount of lost work. Another drawback of check-out approaches is that all objects and subobjects have to be checked-out explicitly whereas in COACT this is transparent for the users.

Like the split/join transaction model [PKH88, KP92], we support the explicit exchange of operations among transactions. The split/join model is, in contrast to COACT, restricted to serializable executions. This limits the possibility to exchange operations between transactions

by means of the provided split and join primitives. Since the model supports only single data copies, access to these data items has to be synchronized, e.g., by a locking protocol, which limits the availability of data. In addition to this, by splitting or joining a transaction the former owner of the operations loses control over these operations. In contrast, in COACT both users can continue their work based on the exchanged results. Another limitation of the split/join model is the lack of higher level operation semantics. The basic split/join transaction model considers only read and write operations. Hence, it can be extended using our merge criteria to be more powerful and flexible in splitting and joining, e.g., merging ongoing transactions, using the semantics of operations and, thus, enabling a closer cooperation between transactions.

The concept of delegation can also be found in ACTA [CR90, CR91, CR92]. Delegation in ACTA means that the responsibility for committing or aborting actions can be delegated from one transaction to another to broaden the visibility of the delegatee and to tailor the recovery properties of a transaction model [CR93, CR94]. How delegation in ACTA can be implemented is described in [MR97]. In COACT both the delegating user as well as the delegatee keep separate responsibilities for the delegated actions. This is possible since our model assumes isolated workspaces where copies of objects reside.

The participant transaction model [Kai90, BK91, Kai95] defines each transaction as a participant in a specified domain. The domain represents the set of user transactions controlled by users collaborating on a common task. Participant transactions in the same domain need not be serializable (only transactions of different domains have to be serializable). Because the participant transaction model is based on read/write accesses to data and no concurrency control applies to a domain, this can lead to inconsistencies of data accessed in a domain (e.g., updates based on data that is no longer valid) and unintuitive behaviour of the system from the user's viewpoint. In COACT, we guarantee that all operation in the history remains legal by exploiting their semantics. The system detects if the exchange of operations would introduce inconsistencies and proposes alternative sets of operations that guarantee consistency of data.

Cooperation in transactional workflow approaches like [ARSS93, SR93, RB94, GH94, GH94, RS95b, TV95, GHM96, PTV97, WR92] is typically characterized by passing results between workflow tasks in a predefined manner. There is no flexibility in passing results back and forth between co-workers. This is essential in non-workflow scenarios, e.g., cooperative authoring and software development processes. In the TRANSCOOP approach, we distinguish between transactional and organizational aspects of a cooperative activity. The former is mainly concerned with consistency of data sharing and exchange and is the focus of the COACT cooperative transaction model. The latter is mainly concerned with the workflow-like coordination of users within a joint effort. The coordination model COCOA deals with both organizational and transactional aspects in a single language without tightly coupling them like in transactional workflows which assign transactional properties to single steps in a workflow. This approach enables more flexible forms of cooperation.

Group-oriented transaction approaches like [KSUW85, FZ89, ELMB92] structure the overall work process by means of a transaction hierarchy and defining application-specific correct-

ness criteria. The cooperative transaction hierarchy approach [NZ90, NRZ92] uses grammars (patterns and conflicts) to define which concurrent executions of sibling transactions are correct from an application's viewpoint. Within CoCoA, one can specify a set of execution rules for workspaces in a cooperative activity. These rules pose restrictions on the order and existence of operations in a workspace history and define termination states. As opposed to [NZ90, NRZ92], we apply these rules only to enforce application-specific correctness of operation sequences within a single workspace and not to define cooperative behavior among workspaces. Moreover, in contrast to [Ska89, NRZ92], the CoCoA execution order rules avoid dead ends caused by interdependencies of different grammars [KTW96b, AKT⁺96, WAN98].

Chapter 4

Support of Mobile Users in CoACT

In this chapter, we show how the CoACT model can be utilized to support cooperation among users in mobile computing systems [KTW97a, KTW97b, KTW97c, Wäs96]. We extend CoACT's system architecture (cf. Section 3.1) in order to adapt it to the special characteristics of mobile environments.

The extensions address new information exchange options, caching of data objects at mobile computers, working in disconnected operation mode, and re-integration of work into the stationary system. We discuss specific problems induced by the possibility of disconnection, and outline how these problems are solved in a mobile CoACT system, thus enabling “disconnected cooperation” among mobile co-workers.

4.1 Mobile Computing

The rapidly expanding technology of cellular communication, wireless local area networks (LAN), and satellite services will make information accessible anywhere and at any time. In the near future, tens of millions of people will carry a personal digital assistant or a portable palmtop or laptop computer. Regardless of size, most of these mobile devices will be equipped with a wireless, modem, or Ethernet connection to the fixed part of the network, and, perhaps, to other mobile computers [IB94, IK95, IK96b]. Moreover, portable computers are now capacious enough to hold “large” databases and provide computational power comparable to a desktop computer, i.e., portable computers can execute general applications such as wordprocessors, spreadsheets, and complex database operations [IK96a, AK93, IB94].

The resulting computing environment, which is referred to as *mobile computing*¹, no longer requires users to maintain a fixed and universally known position in the network and enables nearly unrestricted user's mobility [Sat93, IB94, FZ94, Kat94, IK96b]. Wireless network tech-

¹Sometimes, the terms *nomadic* computing or *wireless* computing are used instead in literature. “Nomadic” emphasizes the fact that users travel around and connect only part-time to the system, whereas “wireless” stresses the fact that radio or infrared technology is used for communication. Since the term mobile computing is more common, we use this term in the following.

nology permits the new generation of mobile computers to act as an integral part of extended distributed computing environments.

4.1.1 Architecture and Characteristics of Mobile Systems

The general, abstract view of a mobile system consists of mobile computers interacting with the fixed network via dedicated mobile support stations. Figure 4.1 shows the existing and widely accepted architectural model of a system that supports mobile computing [IB94, Kat94, WC95, WC97, PS98].

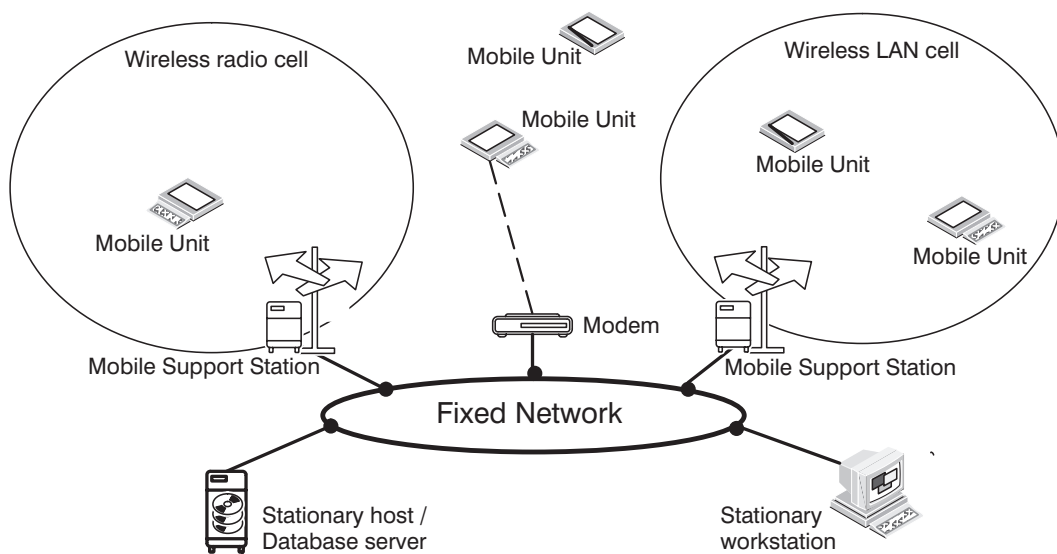


Figure 4.1: General Architecture of Mobile Computing Systems.

The model consists of stationary and mobile components. A *mobile unit* is a mobile computer that is capable of connecting to the fixed network via a wireless link, dial-up, or plug-in connection. Stationary computers are connected via a fixed high-speed backbone network. Some stationary computers act as *mobile support stations*. These computers are equipped with a wireless interface (transceivers) and are capable of connecting with mobile units.

Each mobile support station is in charge of a wireless *cell*. Cells' size can range from picocells (approximately hundred meters in diameter) up to macrocells or even global satellite cells. The mobile unit can communicate with other computers, mobile or stationary, through the mobile support station of the cell in which it resides. Additionally, the mobile support stations may support dial-up connectivity of mobile units, e.g., using analog telephone lines and modems. Current cellular radio technology offers a limited bandwidth of up to 20 Kb/s, whereas current wireless LAN technology offers a bandwidth in the order of 20 Mb/s [PS98, IB94, Kat94].

Table 4.1: Characteristics of Mobile Environments.

Characteristics of wireless communication	Characteristics of mobile units
Frequent disconnections	Part-time switch off and disconnection
Relatively unreliable	Relatively unreliable
Varying and relatively low bandwidth	Limited battery power/life
Asymmetric bandwidth	Frequent location updates
Monetarily expensive (charges)	Subject to loss or theft
Physical support for broadcast	Small user interface

Such a mobile computing environment has different characteristics compared to a stationary distributed system, namely mobility itself, narrow and varying bandwidth, voluntary/involuntary disconnections, and limited battery power. For example, mobile units must operate under a much broader range of network conditions. Mobile units are more prone to loss, destruction, and subversion than static elements. Table 4.1 is based on [PS98] and summarizes the most important characteristics of the mobile units and wireless media with respect to data management. For detailed discussions on these issues, we refer to [IB94, FZ94, IK96a, PS98].

We further elaborate on network disconnections, since they occur frequently in mobile computing and constitute an important issue that has to be addressed in supporting cooperative work among mobile users. Disconnections can be categorized in various ways [IB93, AK93, IB94, Kis96, PS98]. First, disconnections may be *voluntary*, e.g., when the user deliberately avoids network access to reduce cost, power consumption, or bandwidth usage or *involuntary*, e.g., when the mobile unit enters a region where there is no wireless network coverage. Second, disconnections, may be *predictable* or *sudden*. For instance, voluntary disconnections are predictable. Other predictable disconnections include those that can be detected by predicting the battery lifetime or changes in the signal strength. Third, disconnections can be categorized based on their *duration*. Very short disconnections, such as those resulting from crossing cell boundaries (hand-offs), can be masked by hardware or low-level software. Other disconnections may either be handled at various levels, e.g., by the file system or an application program, or may be made visible to the user. Fourth, various degrees of connection (or disconnection) can be distinguished: total disconnection, *weak (dis)connection* (over a low bandwidth channel or an unreliable connection) and strong connection (e.g., using a wireless LAN or plugging into the fixed network). Figure 4.2 illustrates these different kinds of connectivity and the transitions between the different modes.

Of course, disconnections in mobile environments have to be handled differently from network failures that occur in traditional distributed systems. Since disconnections are very common, supporting *disconnected operation* mode, i.e., allowing the mobile unit to operate even when disconnected from the fixed network, is a central design goal in mobile computing. For example, predictable, long-term disconnections can be prepared ahead of time as part of the dis-

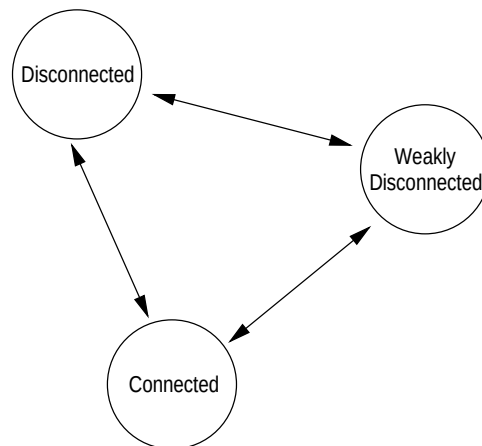


Figure 4.2: Different Kinds of Connectivity in Mobile Environments.

connection protocol. There is a broad consensus that disconnected operation is a key enabling technology of mobile computing [IB93, AK93, IK95, Sat96, Kis96, PS98].

4.1.2 Cooperative Work in Mobile Environments

In recent years, there has been some effort towards eliminating the office and replacing it by a mobile “office”. It appears that progress towards an office-less work environment in a more general setting requires research not only in mobile computing but also in such areas as groupware and workflow [IK95]. Unfortunately, in the past, research in providing computer support for cooperative work has focused only on fixed computer environments with permanent network connections.

Most approaches in the CSCW area have neglected the special characteristics of mobile environments like limited bandwidth or voluntary/involuntary disconnection. Realtime groupware like [EG89, SYZC97, SE98b] and multicast systems like [BC91, BSS91] are only applicable when computers are permanently connected via a fixed high-speed network.

Also in the workflow area, there has been little emphasis on supporting mobile users (notable exceptions are the Exotica/FDMC extension [AGK⁺96] of IBM's FlowMark system and the Information Carriers (INCAs) approach [BMR96]). Supporting disconnected operation in workflow management systems appears to be an intricate task, since workflow execution and disconnected operations have somehow contradictory goals. The execution of workflows involves the coordination of many users, while disconnected operation is geared towards supporting users working (part-time) in isolation.

Especially asynchronous cooperative applications are suitable for being used in mobile environments [KTW97a, KTW97b]. This kind of applications already have to deal with part-time disconnection, one of the central issue in mobile computing. The ability to part-time work in disconnected mode can even be useful when connectivity is available. For example, discon-

nected operation can extend battery life by avoiding transmission of data. Moreover, it can reduce network charges enormously and improves the response-time because operations can be executed locally.

The COACT model provides a framework for long-duration cooperative activities in interactive, multi-user environments. It is well suited for building asynchronous cooperative applications on top of it. COACT uses an “optimistic replication” approach where each user is assigned her own private workspace where copies of the data objects reside. Operations executed by different co-workers do not affect data objects in other workspaces. In order to see the work performed by others, a user explicitly incorporates the operations executed by co-workers into her workspace. This process can be regarded as a (partial) synchronization of different views on the data to re-establish a common view of (parts of) the data. The correctness of the information exchange is ensured by the history merging paradigm.

Optimistic replication is well suited for disconnected operation because it provides much higher availability of data [IB93, AK93, KS92, Kis96]. To avoid locking is of special concern in case of extended disconnections because the possession of locks by disconnected clients would preclude writing and/or reading of other replicas. On the other hand, we have to cope with conflicts caused by concurrent access to different replicas [IB93, AK93, KS92, Kis96].

Conventional approaches to disconnected operation like Coda [KS92, SKM⁺96a, Kis96] use optimistic replication, too, but are based on the assumption that virtually no conflicts occur. Hence, they offer hardly any support for resolving conflicts. This assumption is not valid in cooperative applications which are focusing on sharing data. Thus, it is difficult to apply these approaches to support multi-user cooperation in mobile environments.

In contrast to this, COACT offers a sophisticated merge algorithm. This algorithm reduces the number of conflicts by exploiting the type-specific semantics of operations, thus, reducing the frequency of manual repairs. Additionally, if a conflict cannot be avoided the merger offers different consistent sets of operations. Among these alternatives, the user just has to choose one.

4.2 Mobile CoAct Architecture

In the following, we extend our work on COACT to support disconnected operation mode in mobile environments [KTW97a, KTW97b]. To enable cooperation among mobile users and among stationary and mobile users, we instantiate the COACT system architecture (cf. Figure 3.1) as described below. An overview of the mobile COACT architecture is shown in Figure 4.3. The most important aspect of this architecture is that the private workspace of a mobile user is split into two parts: a stationary one and another part residing on the mobile unit.

The stationary part of the mobile architecture consists of the common workspace of the cooperative activity and one *stationary private workspace* (SPWS) per participating user in a cooperative activity. The stationary workspaces are connected via the fixed network. This stationary part of the architecture is similar to the COACT architecture as described in Section

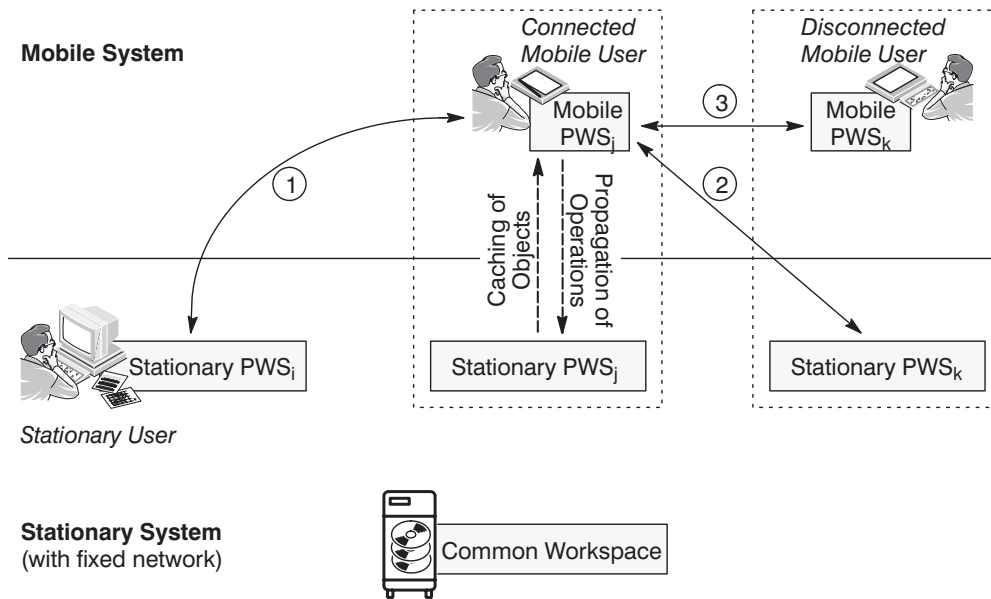


Figure 4.3: Extended COACT Architecture and Information Exchange Options.

3.1. Additionally, we assign each mobile user a *mobile private workspace* (MPWS) that resides at the user's mobile computer. We assume that the mobile computers can part-time connect to the stationary system, e.g., by plugging in the mobile computer into the fixed network, by using a dial-up connection, or by using a wireless link.

According to [PS98], the presented COACT architecture can be classified as a Client-/Intercept/Server model that exploits a messaging and queueing infrastructure for communications of the MPWS with the SPWS. The stationary private workspace is a proxy of the mobile private workspace on the fixed network. From the view of the mobile client, the mobile private workspace appears as the local SPWS surrogate or cache that is co-resident with the cooperative application at the mobile unit.

This architecture alleviates the impact of varying and limited bandwidth, poor reliability, and the possibility of disconnection. For example, the MPWS can be used to satisfy the client's data requirements at disconnection. Similarly, information exchange requests sent to the disconnected mobile user can be queued at the SPWS side and transferred to the client at reconnection. In particular, having both a stationary and a mobile private workspace per user has the following advantages:

- As mentioned, mobile computers have limited resources and not all data objects of a private workspace can be uploaded to the mobile computer. Cache misses (in the MPWS) can be easily handled (if the mobile computer is connected) because the stationary private workspace contains the correct versions of the data objects needed.
- The stationary private workspace is utilized to achieve durability of the modifications to

the workspace state *and* workspace history. Data in the mobile private workspace needs to be persistent (because the mobile unit can be switched off in disconnected mode), but because of the increased risk of a crash and the possibility of theft and loss of the mobile device, data on the mobile computer cannot be considered durable.

- The stationary private workspace serves as a proxy of the mobile private workspace. This enables other users (both stationary and connected mobile ones) to exchange information, i.e., importing and delegating, while the mobile user is disconnected. The importing user may see in the SPWS information not up-to-date, since the user from whom the work is imported is disconnected. Note that in case of a *delegate*, the delegation is queued and the SPWS of a disconnected delegatee is first updated when the delegatee re-connects and initiates the merge.

The initial state of the common workspace, i.e., the state at the beginning of the cooperative activity, is replicated in all stationary private workspaces. For the stationary user the complete content of the workspace, i.e., the data and the workspace history, is reflected in the stationary private workspace. The stationary user can alter the stationary private workspace directly. For the mobile user two questions have to be answered: (1) Where are the data objects? and (2) Where is the workspace history located?

Where are the data objects? The stationary private workspace contains all data objects. All modifications made by those operations which have already been propagated to the stationary system (see below) are reflected in the data objects there, i.e., in the state of the SPWS.

Mobile users execute their operations at the mobile private workspace. This implies that the data objects that are accessed by an operation have to be present in the mobile private workspace. Hence, the user (or the system) first *prefetches* some of the data objects into her mobile private workspace. Due to the limited resources of the mobile computer, only parts of the stationary private workspace may be cached at the mobile private workspace.

A situation where data objects are needed that are not in the MPWS, is called a *cache miss*. While being connected to the stationary system, a cache miss can be easily handled by fetching the required data objects from the stationary private workspace.

How to prefetch the right data objects into the MPWS and how to deal with cache misses in case of disconnection is described in Section 4.4.1 and 4.4.2, respectively.

Where is the workspace history? As described above, a mobile user executes operations only in her mobile private workspace. This applies to both operations directly invoked by the user and operations re-executed due to an information exchange, i.e., a merge of histories. This implies that all history entries are generated at the mobile private workspace.

To keep the proxy of the MPWS at the stationary private workspace up to date and minimize the risk of lost data, we propagate all history entries to the stationary private workspace as soon

as possible and re-execute the operations at the SPWS to achieve equivalent states in both representations of the user's workspace.

In case of connection to the stationary system this can be done immediately. For a disconnected user this is done after re-connection. In case of weak connection, different propagation strategies are possible, e.g., periodically or when the network traffic is low. For details of the re-integration mechanism see Section 4.4.3.

At the stationary private workspace all history entries are stored until the end of the cooperative activity. The storage of history entries at the mobile private workspace can be handled in a more flexible way and depends on the disk space and the autonomy requirements of the mobile user.

To participate in an information exchange while being disconnected, the mobile user needs the complete history of her workspace. This is necessary to compute consistent units of work or to detect all relevant merge conflicts. To undo parts of her work by means of compensation at least all operations succeeding the ones to be compensated have to be available.

If the user needs these facilities only during connection, the history entries at the mobile unit can be dropped after they have been propagated to the stationary system to save disk space on the mobile unit.

4.3 Information Exchange in Presence of Mobile Users

As described in Section 3.1, the COACT cooperative transaction model offers three types of information exchange meta-operations, namely *import*, *delegate*, and *save*. There exist two options for exchanging information among workspaces of co-workers: (1) co-workers can directly exchange working results among their workspaces by means of *import* and *delegate* and (2) users can exchange information through the common workspace by means of *save* and *delegate*.

In the presence of mobile users, we have new options for the information exchange. In the following we describe characteristic examples in detail.

1. *Import* or *delegate* between a stationary user and the mobile workspace of a connected mobile user (cf. exchange option 1 in Figure 4.3).

If parts of the history of the mobile user are delegated to or imported by the stationary user, the necessary operations are taken from the history of the stationary private workspace (not from the MPWS). Since the mobile unit is connected, we can assume that its MPWS history has been entirely propagated to the stationary private workspace.

In the other direction, delegation to the mobile user takes place by making available the respective subhistory at the SPWS of the delegatee. Since the mobile user is in connected mode, she is immediately notified of this delegation and can integrate the corresponding operations into her workspace. A cache miss that might occur while re-executing these operations can be relieved by fetching the object from the stationary private workspace

of the delegatee. An *import* exchange meta-operation of the mobile user is handled in a similar way, without the notification, because the mobile user is the active part.

An analogous approach is used for information exchange between two connected mobile users.

2. *Import* or *delegate* between a connected mobile user and the stationary private workspace of a disconnected (or weakly connected) mobile user (cf. option (2) in Figure 4.3).

Although one of the users is disconnected (or weakly connected) and not reachable, it is nevertheless possible to import from her workspace history. As mentioned above, we use in this case the corresponding SPWS as a proxy of the MPWS. Delegation to the disconnected client is handled similarly as in (1), with the difference that the delegated operations are queued while the delegatee is disconnected (the SPWS of the delegatee is not changed). When the mobile user re-connects, she is notified about the delegation and can then integrate the delegated operations into her private workspace, i.e., merge the histories.

An analogous approach is used when a stationary user instead of a connected mobile user is involved in the information exchange.

3. *Direct import* or *delegate* between two mobile users, both not connected to the stationary system but connected to each other (mobile workgroup [TDP⁺94b]). The direct connection among the mobile units can be established, e.g., using infra-red connection or cellular phones (cf. exchange option 3 in Figure 4.3).

To perform an exchange meta-operation both clients must have access to their complete workspace history, i.e., each client must either be connected to the stationary system or have cached the (entire) workspace history. This is necessary to calculate consistent units of work on the sending client where the source history is located and to detect merge conflicts on the side of the receiving client.

When re-executing or compensating operations during the merge, a cache miss might occur. This situation is similar to cache misses caused by operations directly invoked by a mobile user. If the cache miss cannot be served due to disconnection, the user can decide to interrupt the merge process and confine himself to integrating only those operations that precede the operation causing the cache miss or that are independent of this operation. Another possibility is to abort the merge and integrate no operation at all.

4.4 Disconnected Cooperative Work in CoACT

The idea underlying the support for disconnected (co-)operation in CoACT is as follows. Prior to a (total or weak) disconnection, the mobile unit is in the *data caching* or hoarding state [KS92]. In this state, data objects are (pre-)fetched into the mobile private workspace (cf. Figure 4.4). Caching and prefetching is described in Section 4.4.1.

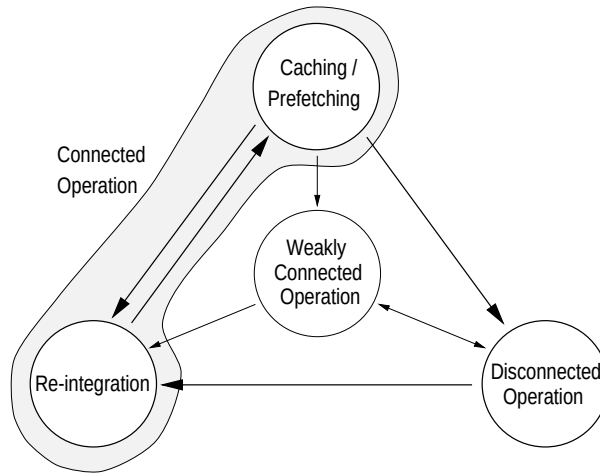


Figure 4.4: Modes of Operation in the Mobile COACT Architecture.

Upon total disconnection, the mobile unit enters the *disconnected operation* state (cf. Section 4.4.2). While disconnected, the mobile unit can use only local data objects available in the MPWS. In contrast to total disconnection, in *weakly connected state* (e.g., due to slow or expensive networks), cache misses can be selectively served if desired. In both states, updates at the mobile private workspace are logged at the mobile unit's stable storage, i.e., the workspace history.

Upon re-connection, the mobile unit enters the *re-integration state* (cf. Section 4.4.3). Re-integration of updates is performed by re-executing the new operations of the MPWS history in the stationary workspace. Propagation of history entries from the MPWS to the SPWS is also possible in weakly connected state. Note, when all operations are re-integrated into the stationary workspace and the mobile unit is connected, both caching and re-integration take place simultaneously since we propagate all operations on the MPWS directly to the stationary workspace.

4.4.1 Caching and Prefetching on the Mobile Unit

To work on the mobile unit, the data objects that are needed to perform the required task have to be in the cache, i.e., the MPWS. While connected to the stationary system, cache misses can be easily handled by fetching the missing data objects from the stationary private workspace (cf. Table 4.2). In weakly connected mode, cache misses can be selectively served if desired.

To minimize cache misses while disconnected or weakly connected, it is crucial to fill the cache with those data objects that are most likely to be needed in the near future. As proposed for the Coda file system [KS92, SKM⁺96a, Kis96], the desired cache content is determined by a mixture of two strategies, namely (1) keeping those data objects in the MPWS the user is currently working with, i.e., applying usual LRU caching, and (2) prefetching certain user-

Table 4.2: Serving Cache Misses and Update Propagation in Different Operation Modes.

	Connected	Disconnected	Weakly connected
Cache miss	Serviced	Non-blocking: raise an “exception”	Different strategies (e.g., serviced, selectively serviced, or not serviced)
Propagation of updates	Immediate	Upon re-connection	Different propagation strategies (e.g., periodic, when network traffic is low, or upon re-connection)

specified data sets, i.e., fetching data objects the user expects to work with in the future. To sustain un-predictable disconnections, prefetching needs to be deployed on a regular basis, e.g., periodically.

With respect to the displacement strategy, we can make use of the behavior-based approach of the COACT model, i.e., we do not need to distinguish between updated and clean data objects in the MPWS. The data objects themselves are *never* written back to the SPWS. Modifications of data objects or the creation of new data objects are communicated to the stationary system by transferring the history entries representing the modifying operations. Therefore, we can remove data objects from the MPWS although they are modified and not yet written back to the stationary private workspace.

A caching strategy for hypermedia authoring. As mentioned, our current demonstrator application for the COACT system is the hypermedia authoring system SEPIA [SHH⁺92, WA95, BWAH96, WA97] (cf. Sections 3.4.2 and 5.3). In the following, we propose an application-specific caching strategy for hypermedia structures in SEPIA.

For the purpose of caching we define a relevance metric on the hierarchical hypermedia network. The relevance of one node to another is determined by their distance in the hypermedia graph, i.e., the number of links or nesting levels that are between both nodes. Nodes that are closer together are considered to be more relevant to each other. The relevance is defined horizontally, i.e., it considers nodes and links within the same authoring space, as well as vertically, i.e., it considers nodes which are contained within other composite nodes.

This relevance metric is used for both LRU caching and user-defined data hoarding. For LRU caching we use the metric to perform prefetching. In case of a cache miss we can fetch not only the required data object but also all other data objects that have a relevance for the missing object above a certain threshold. The value of the threshold is dependent on the current cache contents. At the beginning of the cooperative activity when the MPWS is nearly empty, the value is low, enabling a quick loading of useful data objects into the MPWS. When the cache is filled later on, the threshold is increased. A similar behavior can be used to provide the user

with a simple specification of the data objects she wants to have in the MPWS. She can do this by clicking on certain hypermedia objects. The caching algorithm then determines all other data objects which are relevant to the selected ones.

After determining in this way all hypermedia nodes to be cached, the caching algorithm fetches the connected subgraph containing these nodes, i.e., all links connecting the respective nodes are added. To further optimize the management of disk space on the mobile unit, the user can specify whether she wants to cache voluminous multi-media contents, e.g., audio, video or graphics, that can be contained in atomic nodes of the hypermedia document.

4.4.2 Working in Disconnected Mode

In this section, we describe specific problems that may occur due to disconnection. After fetching the objects from the stationary workspace into the MPWS, the mobile user can work autonomously in disconnected mode. Operations executed on the mobile workspace are logged in the local workspace history of the MPWS.

It is important to recognize that aspects of concurrency control of disconnected mobile clients need *no* special consideration in COACT (as opposed to all other known approaches). It is inherent in the COACT model that conflicts that may occur during disconnected work are resolved later by history merging. This feature of COACT already meets the requirements of multi-user cooperation in mobile environments.

Two major problems have to be addressed in the context of disconnected operation: (1) handling cache misses occurring during disconnection or weak connection, and (2) ensuring uniqueness of operation identifiers and object identifiers within the whole cooperative activity.

While being disconnected, the mobile user can only access data objects that reside within her mobile private workspace. A cache miss occurs when a disconnected user requests the execution of an operation that accesses data objects that have not been prefetched. Cache misses are handled in a non-blocking way in our model. If it is for some reason not possible nor desired to connect to the stationary system, the execution of the operation is aborted and the user is informed about the cache miss. It is up to the user to continue with another task or to re-connect to the stationary system if possible. The same holds if, during weak connection, it is not desired to serve the cache miss.

In case a connection can be established, the cache miss can be handled transparently for the user by sending a request to the corresponding stationary private workspace and fetching the missing data objects (cf. Section 4.4.1). It should be noted that the missing data objects are always available in the SPWS in the required version, because during disconnection no operations are performed in the stationary workspace.

The assignment of operation identifiers to operations requires special treatment in mobile environments. In a COACT architecture which is limited to permanently connected workspaces, uniqueness of operation identifiers can be ensured by a central instance. During disconnected operation it is not possible to obtain a unique operation identifier from the stationary host managing the common workspace.

To ensure the uniqueness of operation identifiers also in case of disconnection, the following approach is taken: Each pair of SPWS and the corresponding MPWS in a cooperative activity is assigned a unique workspace identifier. This workspace identifier is dynamically generated by the stationary system and assigned to SPWS and MPWS when a user joins the cooperative activity (which of course requires a connection to the stationary system). In addition to the workspace identifier each pair of mobile and stationary workspaces maintains a monotonically increasing local operation counter. To ensure uniqueness of operation identifiers among all workspaces, the resulting operation identifier consists of two components: the workspace identifier and the current value of the local operation counter. A similar approach is pursued to generate unique object identifiers (OID) for objects created in a private workspace.

4.4.3 Re-Integration of Work

If a mobile user re-connects to the stationary system, the modifications made in her mobile workspace during disconnection are propagated to the stationary workspace. Note that we always try to keep the SPWS up to date, because it serves as a proxy of the MPWS during disconnection.

Re-integration of work in the CoACT model is based on the re-execution of operations in the SPWS to update its workspace state. Hence, in contrast to other models, it is the history of the operations executed on the mobile workspace that has to be transferred instead of the updated data objects. This can reduce the network bandwidth required for re-integration. The propagation of the MPWS history can be done asynchronously (in the background even in case of weak connections like in [MES95]) and is transparent for the mobile user. Re-execution of the operations on the stationary workspace can be done asynchronously, too, and even if the mobile user is disconnected again.

Propagation of operations to the stationary system *cannot* lead to conflicts because we do not have to merge the histories of the SPWS and the MPWS. The reason is that the MPWS history generated at the mobile unit during disconnection is a *continuation* of the history already reflected at the SPWS and the mobile workspace serves only as a cache of the stationary workspace. If some data objects are newly created in the MPWS during disconnection, their (re-)creation in the SPWS is covered by re-executing the operation that created the object in the mobile workspace. Therefore, we only need to append the MPWS history entries to the history of the SPWS and to re-execute the operations on the objects in the stationary workspace to reflect the modifications done during the period of disconnection.

Propagation of changes to the associated SPWS corresponds to saving the changes on durable storage and can be considered as the default way of re-integration. If, for some reason, propagation to the stationary system is not possible, e.g., due to a network failure on the fixed network or because the machine hosting the SPWS is down, the model still allows to apply the CoACT information exchange meta-operations in order to make results visible to cooperating users (see exchange option (3) in Figure 4.3). In critical situations, this can be an alternative to the re-integration described above. This opportunity allows to distribute changes carried out

during disconnection to other participants of the cooperative activity before they are installed in respective SPWS. Although re-integration into the SPWS is necessary in any case, the results achieved in the MPWS can become earlier visible to co-workers, e.g., in case of mobile workgroups, which increases the flexibility of the model.

Optimizations to reduce the size of the history to be transferred during re-integration are possible. If an operation has been compensated due to a prior merge conflict or explicitly by a user, both the original operation and its compensating operation can be eliminated from the history. The reason is that they semantically correspond to a null operation. Moreover, optimizations to the re-execution process are possible (cf. Section 3.5.4.3). For instance, as for merging histories, it is not required to (transfer and) re-execute transparent operations at the SPWS.

4.4.4 Failure-Resilience

The diversity introduced by mobile computing brings new challenges to failure-resilience. Types of failures that are very rare in fixed environments are more likely in mobile environments, e.g., physical damage becomes more probable. Mobile computers can be lost or stolen. Failures due to power problems can be frequent events.

Failures on a mobile computer can be classified in two categories [NF97]: (1) permanent failures that cannot be repaired autonomously by the mobile computer and (2) transient failures that temporarily affects the functionality of the mobile device.

In our model, recovery of transient failures is addressed with traditional database recovery concepts. Before an operation is executed, the corresponding history entry is written to persistent storage (write-ahead-logging) at the mobile unit. Furthermore, the initially cached workspace state is stored on disk at the mobile unit (checkpoint). In case of a transient failure the last state of the mobile private workspace can be recovered by re-executing the logged operations on the checkpointed workspace state. This process can be optimized by periodically checkpointing the modified data objects on stable storage of the mobile unit.

Permanent failures are handled using the stationary workspace as a reliable backup of the corresponding mobile workspace. Recoverability of the stationary workspace is achieved in the same way as for the mobile private workspace. Operations executed on the mobile unit during disconnection are immediately propagated to the stationary workspace if possible. Thus, the state of the mobile workspace can be recovered, and, hence, is durable up to the last propagated operation.

4.5 Comparison with Related Work

Up to now, most research with respect to disconnected operation has been done in the area of distributed file systems. Coda [KS92, SKM⁺96a, Kis96] which is based on the Andrew File System (AFS) is the first implementation of a file system explicitly supporting disconnected

operation. The system uses an optimistic replication policy. Before a mobile client disconnects, it hoards useful data in its cache based on a LRU policy and hoard profiles. A disconnected client can continue to work by using any data in the local cache. If a cache miss occurs, the operation fails and an error code is returned. When the client re-connects, the updates are integrated. In case of conflicting file updates, the user manually has to resolve the conflict. Support for weak connectivity in Coda is described in [MES95].

The approach of LittleWork [HH93] which adds disconnected operation to AFS is similar to that used in Coda. During disconnection the mobile client can perform local operations whereas network operations are logged. If a connection appears, the log is transferred to the stationary system and replayed.

Ficus [HPGP92] is a replicated file system supporting disconnected access, too. Ficus adopts a primary copy approach to replication control. Only a site with a write token can modify the file. Thus, a disconnected client has to have the write token (primary copy) to modify the file. At re-connection, communication to the secondary copies has to be established.

All these systems suffer from several problems: One big problem is that they have only a poor notion of dependencies among multiple data objects and user interactions, i.e., the operations executed on the data. In traditional file systems, such dependencies are generally not recorded, and appropriate compensation operations are not explicitly specified. For example, Coda and Ficus deal only with write/write conflicts. The problem of read/write conflicts between files is partially addressed by so-called isolation-only transactions [LS94]. However, it is not clear how a transaction reacts in case of a cache miss. Even in case of a connection, the missing file might have been modified by another client and the right version of this file is no longer available to resolve the cache miss.

Moreover, all of the systems mentioned above do not consider type-specific semantics of operations executed during disconnection (with an exception of the directory operations in Coda), thus, the probability of conflicts is high. Therefore, these systems can only be used when the degree of data sharing is very low which is obviously not true in cooperative work environments. One exception is Rover [JdT⁺95, JTK97]. In this approach operation semantics is used to reduce conflicts. However, this model sacrifices the independence of the merge functionality and the application. In Rover the information about operation semantics and the algorithms for the integration of changes are realized within the cooperative application. This increases the specification overhead and makes it impossible to use pre-existing applications which cannot be modified.

Less work has been done in the field of database systems to support disconnected clients. One exception is the proposed extension of Thor [GKLS94] to support disconnected mobile clients. Thor is a client/server object-oriented DBMS, where each client caches copies of the persistent data objects residing at the server site. In comparison to the file systems described above, special emphasis has to be put on the caching of data objects before disconnection because data objects in object-oriented DBMS are smaller than files and might be heavily interconnected. Additionally, computations in a DBMS have to run within ACID transactions.

The authors of [GKLS94] propose to use the high-level semantics of data objects to decrease the number of conflicts and, thus, the number of transaction aborts at re-connect. This is similar to our approach. However, as serializability is used as the correctness criteria in Thor, cooperative multi-user activities are not supported. Instead, it was proposed that data sharing among multiple users is controlled by the application itself, i.e., outside the database system.

The problem of disconnected cooperation among multiple users is addressed in [SK93] for the special case of software engineering environments. The system described is based on a file repository which includes additional control mechanisms. In contrast to the approaches described so far, it uses a check-outout/check-in model where files have to be locked before being cached at the mobile clients. Two new lock modes are introduced: dirty-read and generated-exclusive. The first mode allows users to read inconsistent data, whereas the latter is used for files like executables that can be re-generated by special tools, e.g., “make”. Generated-exclusive files are treated differently than exclusively locked files during the re-integration process. The reason is that they can be re-generated in their newest version from other files. This approach seems to be useful for this particular application domain, but obviously, it is not applicable in the general case.

The Bayou system [DPS⁺94, TDP⁺94b, TDP⁺94a, TTP⁺95] aims at supporting data sharing among mobile users. Bayou is based on replication of data on several servers. Each client requires at least access to one server to perform useful work. Bayou servers propagate writes among themselves during pair-wise contacts, called anti-entropy sessions. This corresponds to a read-any/write-any replication model. Application-specific conflict detection is accomplished through the use of dependency checks. Each write operation in Bayou includes a dependency check consisting of an application-supplied query and its expected result. In case of conflicts, i.e., the actual and the expected results differ, an application-defined merge procedure is invoked. Although Bayou allows application-specific conflict resolution strategies, the specification effort required is enormous. For each write operation application-specific dependency checks and merge procedures have to be specified which are difficult to validate against the application semantics. In contrast to that, COACT gives the opportunity to specify exactly the application semantics required for conflict detection and resolution in the merge relations.

In some other systems, techniques for semantic-based conflict detection and resolution have been proposed, too. As mentioned above, Coda provides extended mechanisms to resolve conflicting directory operations. An extension of Coda to resolve file system conflicts by application-specific resolvers was proposed in [KS95c]. This approach suffers from the same problems as the Bayou system. Oracle's symmetric replication product [Dow95] also includes the notion of application-selected resolvers for relational databases. Other systems, like Lotus Notes [Lot96, Lot97] do not supply application-specific mechanisms to handle conflicts, but rather create multiple versions of data objects when conflicts arise.

Chapter 5

Prototype Implementation of CoACT

In this chapter, we describe the prototype implementation of the CoACT cooperative transaction model within the context of the TRANSCOOP demonstrator system [dBLP⁺96, WEF⁺96, KE98]. As proposed by the TRANSCOOP reference architecture [dBLP⁺95b, dBLP⁺95a, LdBTW98], the demonstrator consists of a specification environment, providing means for the specification and verification of cooperative activities, and a runtime environment, offering transactional support for cooperative applications.

The TRANSCOOP runtime environment is implemented as an extension of the object-oriented DBMS VODAK [DIM95] developed at GMD-IPSI, i.e., all cooperation facilities are supported as DBMS services. The main module of the TRANSCOOP runtime environment is the CoACT cooperative transaction manager that implements the history merging mechanism and CoACT's exchange meta-operations. Other important extensions of VODAK with respect to the CoACT model address the realization of the workspace concept and the management of workspace histories.

The remainder of this chapter is structured as follows. First, we give an overview of the TRANSCOOP demonstrator system. Then, we present the VODAK system and the SEPIA hypermedia authoring application used for demonstration purposes. Afterwards, we describe the prototypical implementation of the CoACT model in more detail. We report on the experience we made during the system design and describe how the new functionality is presented to the users of the selected demonstrator application.

5.1 Overview of the TRANSCOOP Demonstrator System

For the implementation of the demonstrator system, we chose a subset of the concepts developed in the TRANSCOOP project, that was, on the one hand, powerful enough to demonstrate our approach to cooperation, and on the other hand, allowed a reliable implementation within the time frame of the project [dBLP⁺96]. The implemented parts include most of the tools of the specification environment, the CoACT transaction model, as well as an enforcement mechanism for step constraints of CoCoA [WEF⁺96]. The resulting demonstrator system allows us

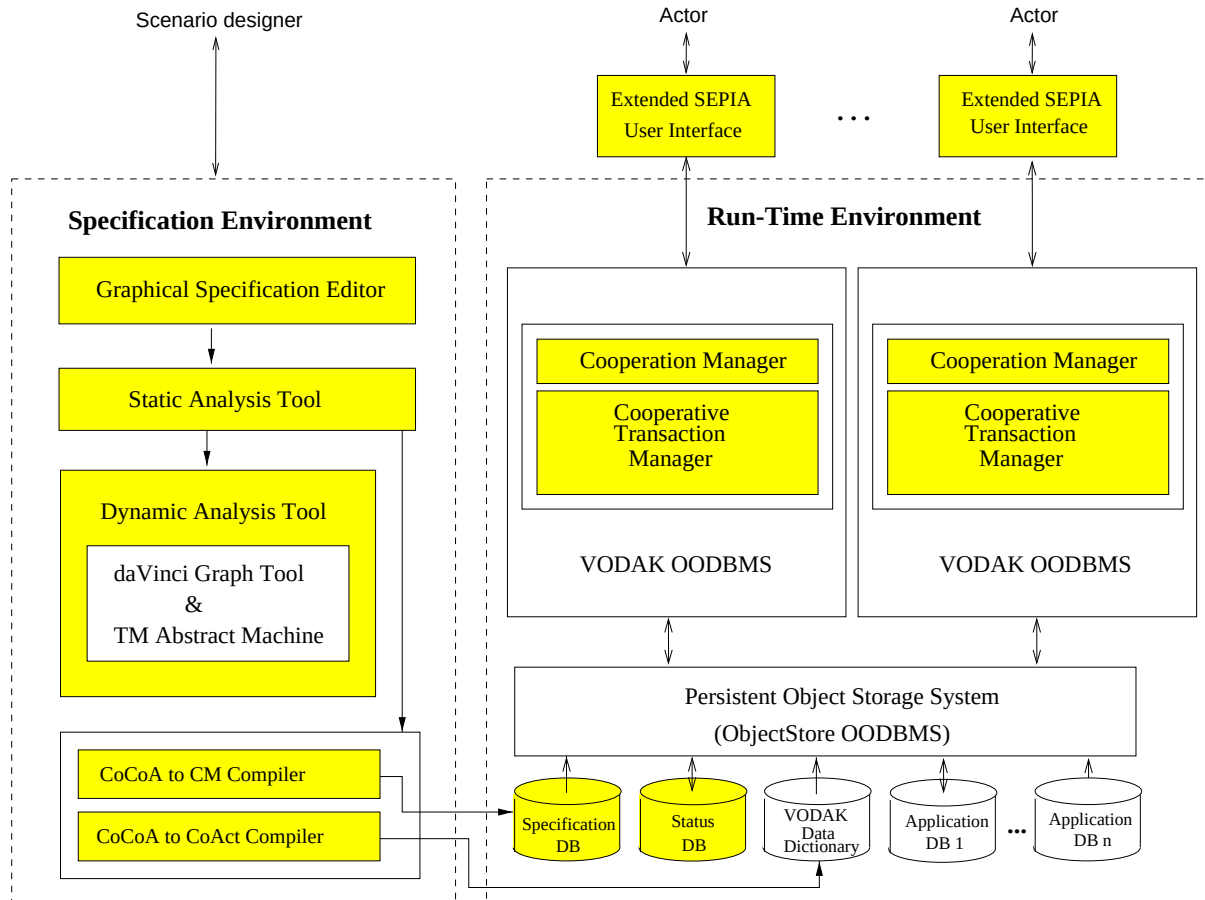


Figure 5.1: Architecture of the TRANSCOOP Demonstrator System.

to show how cooperative activities are specified, validated, compiled for execution, and finally executed [AEF⁺97].

Figure 5.1 shows the overall architecture of the demonstrator system [dBLP⁺96, WEF⁺96, KE98]. The grey-shaded components in the figure indicate the software modules developed and implemented in TRANSCOOP; the white components are pre-existing software modules.

According to the TRANSCOOP reference architecture [LdBTW98, dBLP⁺95b, dBLP⁺95a], the demonstrator system consists of two major components: the *specification environment* and the *runtime environment* [dBLP⁺96, WEF⁺96]. The specification environment provides all tools needed by a designer to create specifications in the CoCoA language (cf. Section 3.6.1). The specification environment contains a graphical specification editor, a parser/type-checker, tools for analyzing and simulating CoCoA specifications, and two compilers that turn the specifications into representations used by the runtime environment.

The demonstrator runtime environment has a transaction management support system as its central component that enables the execution of cooperative activities described by a CoCoA specification. Remember that two different views of a cooperative activity come together in

CoCoA: the *organizational* view and the *transactional* view. We decided to reflect these two views in the demonstrator runtime structure. Thus, the transaction management support system of the runtime environment has separate modules: (1) a *cooperation manager* (CM) [dBLP⁺96, WEF⁺96] that governs and coordinates the cooperation during an execution of a cooperative activity and (2) a *cooperative transaction manager* (CTM) [KTW96a, Sal96, WEF⁺96] that ensures the transactional correctness of a cooperative activity. The CTM implements the full functionality of the CoACT cooperative transaction model.

The runtime environment also contains a *persistent object storage system*, which is used to store compiled CoCoA specifications, runtime status information for the CM and CTM, VODAK-specific data, and application-specific data [dBLP⁺96, WEF⁺96]. The storage system we use in the demonstrator is ObjectStore [LL91]. The extended *graphical user interface* (EGUI) [WEF⁺96, Sar98] enables end-users to interact with the TRANSCOOP demonstrator system.

In the following, we describe the overall functionality of the TRANSCOOP demonstrator system by “walking through” its components to highlight their use and dependencies. For more information about the TRANSCOOP specification environment, we refer the reader to [EFdBP96, FEdB98, KE98, WEF⁺96, dBLP⁺96]. The runtime environment is described in more detail in Section 5.4, with the focus on the implementation of the CoACT model.

1. Specification Environment.

- (a) *Graphical Specification Editor*: A designer uses the graphical editor to build and structure a CoCoA specification. The graphical specification editor is based on a graphical modeling method called GRACOOP (Graphical Language for Cooperative Processes) [EFdBP96]. It allows the specifier to model the organizational structure of a cooperative activity as a graph. Non-graphical information is specified using dialog boxes and editing windows; this information annotates the graphical elements of the step diagram. There is also a *matrix tool* supporting the specification of commutativity relations.

The graphical editor has been implemented using the MetaEdit+ Method Workbench, which supports the development of graphical modeling methods and tools [Met96]. A back-end to the graphical editor generates a textual representation of the specification, which is stored in the file system, where it can be accessed in later stages of development.

- (b) *Static Analysis Tool*: This tool performs static analysis checks on the CoCoA specifications stored in the file system. At its core lies a parser/type-checker. The tool checks a number of context-sensitive requirements related to the definition and use of data operations and communications within a CoCoA specification.

The CoCoA static analysis tool is implemented in C++; the software extends an already existing parser/type-checker for TM [BBdB⁺96]. The static analysis tool

implements several back-end output generators. The back-ends include two compilers to the runtime environment, and a mapping to a TM schema [BBdB⁺96] suitable for simulation using the dynamic analysis tool.

- (c) *Dynamic Analysis Tool*: This tool is used to simulate the organizational behavior of a CoCoA specification using the *TM Abstract Machine* [FB96]. It offers facilities for the study of the execution properties of a CoCoA specification. To do this, the specification is mapped to a TM schema [BBdB⁺96] that defines data structures and operations for storing and manipulating meta-information about the dynamic state of the cooperative activity. During simulation, the activity states are displayed using the *daVinci Graph Visualization Tool* [FW94b, FW94a].

- (d) *CoCoA to CM Compiler*: This compiler maps the organizational aspects of a CoCoA specification into a data structure that is used by the cooperation manager of the runtime environment to manage the organizational constraints within the cooperative activity. These data structures take the form of an *object network*, which captures the organizational aspects of a CoCoA specification.

The compiler retrieves the CoCoA sources from the file system and stores the object network in the *specification database* [dBLP⁺96, WEF⁺96] for later interpretation by the cooperation manager. The CoCoA to CM compiler is described in detail in [WEF⁺96].

- (e) *CoCoA to COACT Compiler*: This compiler maps those parts of a CoCoA specification that are needed by the COACT transaction model into input specifications for the cooperative transaction manager.

The compiler retrieves the CoCoA sources from the file system and generates two output files for a given specification: one contains the conflict definitions among invocable operations, the other contains the specification of the compensation function. These files are inserted into the *extended VODAK data dictionary* and are used by the cooperative transaction manager during the execution of a cooperative activity.

2. Run-time Environment.

- (a) *Specification Database*: The specification database [dBLP⁺96, WEF⁺96] contains the compiled version of the organizational aspects of a CoCoA specification. It is implemented using the object-oriented DBMS ObjectStore [LL91].
- (b) *Status Database*: The status database [dBLP⁺96, WEF⁺96] stores administrative information to support an execution of a cooperative activity, such as the workspace histories of the users participating in the cooperative activity and the runtime structure of the step specification. The status database is implemented using the object-oriented DBMS ObjectStore [LL91].

- (c) *Cooperative Transaction Manager (CTM)*: The CTM implements the basic functionality of the COACT cooperative transaction model. It maintains workspace histories, provides the information exchange facilities, computes dependencies and conflicts between operations, and implements the history merging algorithms. The cooperative transaction manager is described in more detail in Section 5.4.
- (d) *Cooperation Manager (CM)*: The CM monitors and enforces the organizational aspects of a cooperative activity. It verifies that each requested data operation, information exchange meta-operation, or communication is in fact permitted by the CoCoA specification at the particular time when it is requested. For details on the implementation of the cooperation manager, we refer the reader to [WEF⁺96, dBLP⁺96].
- (e) *Extended Graphical User Interface (EGUI)*: To support user interaction in the exchange of results, this interface extends the SEPIA graphical user interface with facilities for viewing and manipulating workspace histories, such as selecting operations for exchange and resolving merge conflicts. The extended graphical user interface is described in more detail in Section 5.4.5.

5.2 The VODAK Object-Oriented DBMS

As the VODAK object-oriented DBMS [DIM95] developed at GMD-IPSI provides the basic platform for both the prototype implementation of the COACT transaction model and for our demonstrator application, its relevant concepts are introduced briefly in the following.

The VODAK system comprises an object-oriented data model, an object-oriented modeling and programming language (VML) [DIM95, KAN94], an object-oriented query language (VQL) [AF95], multimedia database extensions [RNL95, Bol95, BKL96, TK96], as well as sophisticated transaction management [MRW⁺93] and distribution facilities [KFM⁺96].

5.2.1 VODAK's Data Model

From the data modeling viewpoint, VODAK includes not only standard features of object-oriented DBMS (such as object identity, encapsulation, complex data types, inheritance and polymorphism), but also advanced modeling concepts that have been designed to provide better support for the integration of heterogeneous information resources [KFM⁺96, KAN94, KFA94]. These are (1) *strict separation of types and classes* [GPN91] to clearly distinguish extensional from intensional aspects and (2) *meta-modeling* capabilities [KS95a]. Meta-modeling allows to extend the data model with new data modeling constructs, like semantic relationships, which are usually hard-coded into database management systems.

The VODAK Modeling Language (VML) consists of a DDL part which is used to define database schemas, and a procedural, computationally complete DML part for the implementation of database schemas. A comprehensive presentation of VML can be found in

[DIM95, KAN94]. A concise characterization of VML is as follows. VML distinguishes between *data values* and *objects*. Each data value is out of a domain determined by its *data type*. Each object belongs to a domain constituted by a *class*. Objects exist persistently, i.e., they exist independently of the scope of an application program, whereas data values are transient. Objects are identified through unique logical *object identifiers* (OID), whereas values of data types have no identifier.

All instance objects of a class share the description of their *interface*, the structure of the object's *state*, and their *implementation*. This description constitutes the type of an object and is defined through *object type* definitions. The interface consists of a set of methods, more precisely a set of *method signatures*. The state of an object is constituted by a tuple of data values according to the property definitions of the object type. Properties of objects may be of primitive or complex data type, or may contain object references (OID values). Access to the object's state is possible only via the execution of methods in the interface. *Class definitions* determine the domain for the classes. They refer to object types and define class objects that model the containers of objects, i.e., the *extensions* of the classes. A *database schema* consists of class definitions, data type declarations, and object type declarations.

In VODAK, classes are first-class objects, i.e., objects themselves. They are instances of classes which are called *metaclasses*. Metaclasses are used in VODAK to describe the common behavior of *application classes* and their instances, thus, tailoring the VODAK data model to the needs of particular application domains like hypermedia document storage [BA94, WA95, BAN95, WA97]. VODAK's meta-modeling capabilities were heavily used in the implementation of the SEPIA database schema (cf. Section 5.3.2).

5.2.2 Architecture of the VODAK System

The VODAK system has a standard client-server architecture [DIM95], where the VODAK server can rely on different storage managers. The version of VODAK we used was based on the C++-based object-oriented DBMS ObjectStore [LL91]. Client applications to VODAK are written in a programming language for which an application programming interface is available (currently C++ [LBL95] and Java [BW96]).

In the following, we give a brief overview of the internals of VODAK Version 4.1. VODAK provides a complete database management system kernel including a data dictionary, a message handler, a query optimizer and executor, an object manager, a shared object buffer, and a transaction manager responsible for concurrency control and recovery. Figure 5.2 illustrates the internal architecture of VODAK. Note that the grey shaded modules in Figure 5.2 run within a single operating system process (one process per concurrent user). The single VODAK processes are themselves clients to the ObjectStore system. Multi-user mode in VODAK is realized using a shared object buffer (implemented as UNIX shared memory segments) that is shared among all processes of a VODAK server instance. In the following, we describe the different components of the VODAK system.

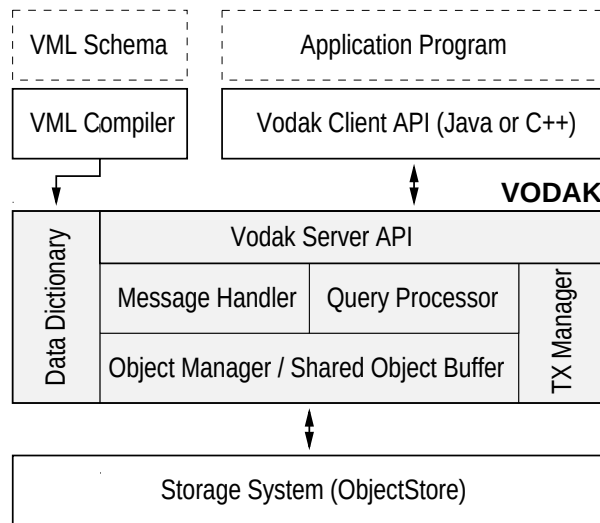


Figure 5.2: Simplified Architecture of the VODAK Object-Oriented DBMS.

- *VODAK Compiler*: The VODAK compiler performs syntactic and semantic analysis of database schemas written in VML. It generates codes for methods and inserts schema information into the data dictionary.
- *Data Dictionary*: The data dictionary provides a complete description of VML schemas expressed within the VODAK data model. Since the data dictionary is a VODAK database itself, the schema information is accessible through VML method calls and VQL.
- *VODAK Server API*: The server application programming interface is responsible for connection handling and communication with application programs implemented on top of one of the VODAK client APIs.
- *Message Handler*: The message handler provides the call-level interface for the server API to VODAK databases. The message handler receives method calls consisting of the name of the method to be executed, the receiver object's OID, and the arguments of the method call. Its main task is to resolve and dispatch these method calls at runtime, and to execute the compiled code.
- *Query Processor*: VQL queries are algebraically optimized by a rule-based query optimizer. The resulting physical query execution plans are processed by the VQL query executor.
- *Object Manager and Shared Object Buffer*: The task of the object manager is to handle accesses to VML objects, i.e., dynamically load objects from the storage system into the

shared object buffer, replace objects in the shared object buffer and write them back to the storage system when necessary.

- *Transaction Manager*: The transaction manager implements different strategies for transaction management, including the open nested transaction model [WS92, MRW⁺93, MR93]. In flat transaction mode, transactional commands are simply forwarded to the storage system.

5.3 The Demonstrator Application: SEPIA

In this section, we describe the implementation of the hypermedia authoring system SEPIA that was used as our demonstrator application.

In our overview of the SEPIA system, we focus on the implementation variant that is based on VODAK. Basically, this means that the SEPIA system is modeled as a VML database schema. Modeling the operation types available in SEPIA within a VML schema allows to exploit their high-level semantics within the COACT cooperative transaction model. In our description, we abstract from the implementation details of the VML schema and limit ourselves to a general description of the system functionality and its architecture.

5.3.1 Overview of the SEPIA Hypermedia Authoring System

A hypermedia document in SEPIA consists of nodes and connecting links, which enable authors to create and present associative reference structures within and between documents. The nodes can contain multimedia information, e.g., plain text, graphics, pictures, and audio. The basic functionality of SEPIA is provided via the concept of activity spaces that support the different activities of the hypermedia authoring process, i.e., planning, content acquisition, development of an argumentation structure, and preparation of the final document [SHT89] (see also Section 3.4.2).

The original SEPIA system and its user interface were implemented in SmallTalk-80 [SHT89]. Because the demonstrator system is based on VODAK, we have chosen the VODAK-based implementation of the SEPIA system [WA95, BWAH96, BLW96, WA97] as the demonstrator application.

Figure 5.3 illustrates the overall client-server architecture of this system. The *SEPIA backend* is implemented as an extension of VODAK and is responsible for the persistent management of SEPIA hypermedia structures. The *SEPIA frontend* provides the functionality of the SEPIA backend to the user via a graphical user interface. The SEPIA frontend is implemented in Java [AG98].

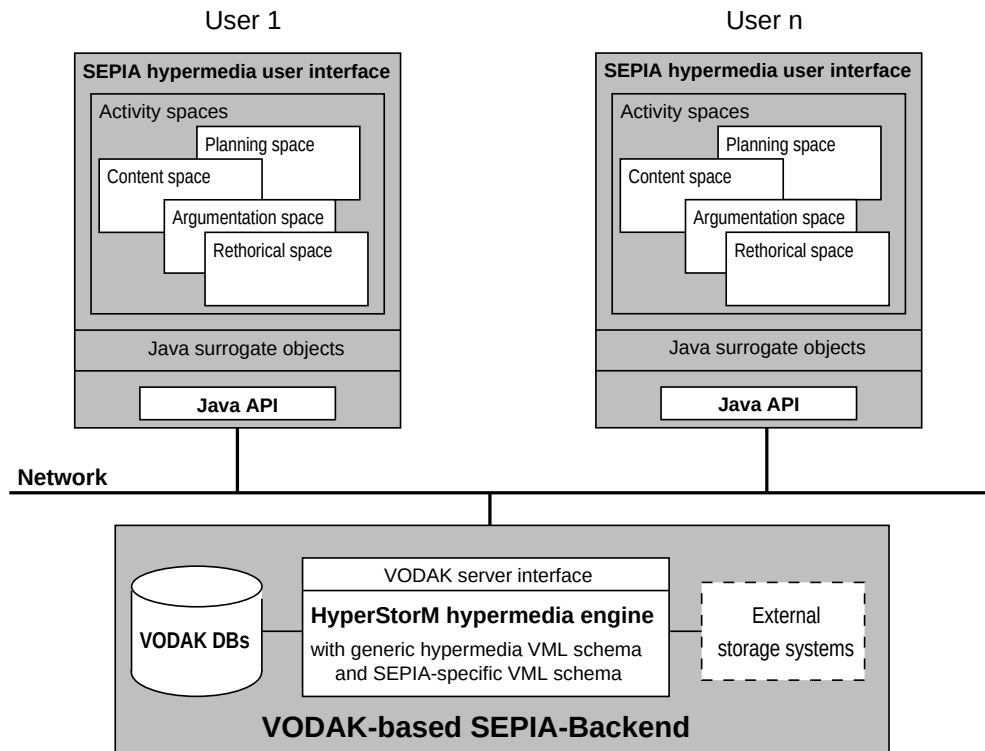


Figure 5.3: Client-Server Architecture of the VODAK-Based SEPIA Implementation.

5.3.2 The VODAK-Based SEPIA Backend

Hypermedia applications need generic support for creation, persistent storage, manipulation, composition, and retrieval of hypermedia documents. Such generic functionality can be provided by an extended DBMS, usually called a *Hypermedia Engine* [WA95, WA97, BWAH96]. In the HYPERSTORM project, we implemented such a hypermedia engine based on VODAK, utilizing the extensible data model and the multimedia extensions of VODAK.

To that end, first a basic set of hypermedia document modeling primitives, including atomic content objects, composite objects, different kinds of hypertext links, and activity spaces, were developed using the meta-modeling approach of VODAK. The result was a set of VML meta-classes implementing a generic hypermedia document model [WA95, WA97, BWAH96]. Figure 5.4 sketches some methods available for `HME_Node_MC`, the generic metaclass for nodes. A subset of the methods implemented for the hypermedia metaclasses constitute the invocable operation types of the SEPIA *CAT* outlined in Section 3.4.2.

This modeling approach allowed to move a substantial part of the hypermedia application semantics into the VODAK system. Having the operation semantics available within the VML database schema was an important prerequisite for enabling cooperation by the COACT model at the level of complex operations.

Using these hypermedia modeling primitives the SEPIA document model was instantiated.

```

DATATYPE CHANGE_INFO_ACONTENTS = [composite: OID, inComposite: OID];
DATATYPE CHANGE_INFO_NAME = [inComposite: OID, oldName: STRING];
DATATYPE CHANGE_INFO_POSITION = [inComposite: OID, oldPos: POINT];

CLASS HME_Node_MC METAClass Metaclass
  INSTTYPE HME_Node_InstType
  INSTINSTTYPE HME_Node_InstInstType
END;

OBJECTTYPE HME_Node_InstInstType
...
INTERFACE
METHODS
  changePosition(pos: POINT): CHANGE_INFO_POSITION;
  changeName(aName: STRING): CHANGE_INFO_NAME;
  createAtomicContent(aClass: OID, name: STRING): CHANGE_INFO_ACONTENTS;
  removeAtomicContent(atomicContent: OID): OID;
  openAtomicContent(atomicContent: OID): BYTESTRING READONLY;
...
END;

```

Figure 5.4: Sample Interface of the Metaclass HME_Node_MC.

This was done by declaring several VML applications classes (specific to SEPIA) as instances of the generic hypermedia metaclasses. These application classes represent the different node and link types (including their constraints) available in SEPIA. Figure 5.5 sketches some of these application classes. For example, the `ArgumentationSpace` can contain nodes of the classes `Datum` and `Claim` and links of the class `Supports` which are allowed to connect instances of `Datum` with `Claim`, `Claim` with `Claim`, but not `Claim` with `Datum`. For more details on the implementation of the VML database schema for SEPIA, we refer the reader to [WA95, BWAH96, WA97, WEF⁺96, dBLP⁺96, Sar98].

5.3.3 The SEPIA Frontend

To provide the end-user authors with access to the hypermedia data, a graphical user interface (GUI) to the SEPIA backend has been implemented. This SEPIA frontend visualizes the hypermedia data as stored in the database and allows the user to navigate through and modify the SEPIA hypermedia documents.

To give the reader an impression of the functionality of the SEPIA system available to authors, we describe in the following the graphical user interface of the SEPIA system (cf. Figure 5.6). SEPIA supports the hierarchical structuring of documents. The top level consists of (logical) databases. Each database is divided into authoring projects that consist of four authoring-specific activity spaces. These are made available to the user via the small windows shown on the left side in Figure 5.6.

SEPIA activity spaces are represented by separate windows for different tasks within each authoring project. For example, the right part of Figure 5.6 shows the space browser visualizing the planning space of the authoring project “`DemoProject`”. Task-specific node types, link

```

CLASS ArgumentationSpace METAClass HME_CompositeContent_MC
  INIT
    //Nodes:
    ArgumentationSpace->addSubGroupClass(Datum);
    ArgumentationSpace->addSubGroupClass(Claim);
    ...
    //Links:
    ArgumentationSpace->addSubGroupClass(Supports);
    ...
  END;

CLASS Datum METAClass HME_Node_MC
  INIT
    Datum->addElementClass(TextContent);
    Datum->addElementClass(ImageContent);
    ...
  END;

CLASS Claim METAClass HME_Node_MC
  INIT
    Datum->addElementClass(TextContent);
    Datum->addElementClass(ImageContent);
    ...
  END;

CLASS Supports METAClass HME_DirectedLink_MC
  INIT
    Supports->addLinkReference(Datum, Claim);
    Supports->addLinkReference(Claim, Claim);
    ...
  END;

```

Figure 5.5: Sample Application Classes of the VML Database Schema for SEPIA.

types, and content types are available within these activity spaces. These types are implemented in the database schema as different application classes. The available node and link types in a specific activity space are displayed in a palette on the left-hand side of the browser. The palette is also used to create new nodes and links.

Depending on their type, nodes have different menus to create and access their contents. Nodes in SEPIA can be either composite or atomic. Composite nodes of the hypermedia data model represent subspaces similar to the activity spaces mentioned above. These are visualized as a separate space browser in the GUI. For atomic nodes, different content types (text, audio, etc.) can be created, depending on the content types defined for the respective node type, as specified in the database schema.

The operations offered by the SEPIA database schema are invoked by the GUI using the call-level interface of the Java Client API to VODAK [BW96]. For example, if a user requests to open a composite node (by selecting an “open” command from the node's pop-up menu) a corresponding method call is sent to the SEPIA backend and executed there. The SEPIA backend then returns information about the contents of the composite node. Using the information returned, the GUI can present the contents of the opened composite node in a browser to enable

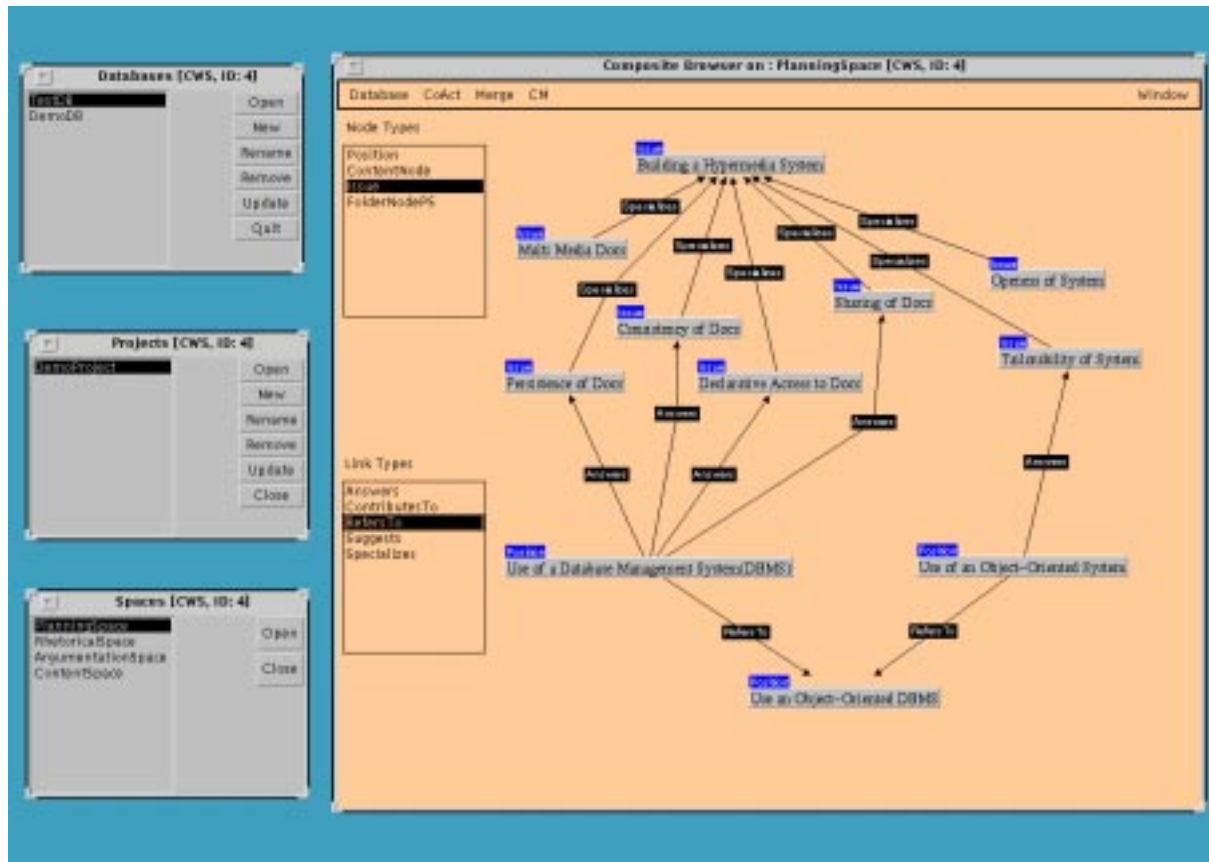


Figure 5.6: The Graphical User Interface of the SEPIA Frontend.

the user to access and manipulate the data. Further details on the implementation of the SEPIA frontend can be found in [Sar98, BLW96].

5.3.4 Extensions of the VML Database Schema Needed for COACT

As described in Section 3.4.1, a COACT cooperative activity type consists of the invocable operation types, a compensation function that specifies the compensation operation for every invocable operation, and the commutativity relations among the invocable operations of the *CAT*.

In our demonstrator application, we consider all VML methods (of the hypermedia meta-classes) that can be invoked by the SEPIA graphical user interface to be invocable operation types. These operations are explicitly specified in the CoCoA specification for SEPIA [WEF⁺96]. The commutativity relations and the compensation function are specified in this CoCoA specification, too. Their compiled versions are used at runtime by the cooperative transaction manager to ensure the correctness of history merging. Note that in our demonstrator application, commutativity relations and the compensation function need only to be provided

```

CLASS HME_Node_MC METAClass Metaclass
  INSTTYPE HME_Node_InstType
  INSTINSTTYPE HME_Node_InstInstType
END;

OBJECTTYPE HME_Node_InstInstType
...
INTERFACE
  METHODS
    changePosition(pos: POINT): CHANGE_INFO_POSITION;
    ...
    // added for CoAct cooperative transaction model:
    comp_changePosition(pos: POINT, returnValue: CHANGE_INFO_POSITION);
    comp_changeName(aName: STRING, returnValue: CHANGE_INFO_NAME);
    comp_createAtomicContent(aClass:OID,name:STRING,
                           returnValue: CHANGE_INFO_ACONTENTS);
    comp_removeAtomicContent(atomicContent: OID, returnValue: OID);
    ...
END;

```

Figure 5.7: Sample Metaclass Extended with VML Methods for Compensation.

for the methods of the generic hypermedia metaclasses, not for each application class separately. This was achieved by a careful application design and has decreased the specification overhead enormously.

In our prototype, the VML methods for compensation must be explicitly specified in the VML database schema. The cooperative transaction manager invokes these methods in case compensation is required according to the compensation functions specified in the COCOA specification. Thus, we extended the generic hypermedia metaclasses by VML methods for compensation. Figure 5.7 sketches some of the methods added for this purpose.

An alternative would be to specify the compensation function directly in the VML database schema, using the UNDO clause offered by the implementation of the open nested transaction model within VODAK [DIM95, KFM⁺96]. However, this assignment would be rather static, and the compensation function could not be analyzed and used for simulation purposes in the TRANSCOOP specification environment. Therefore, we decided to include the compensation function in the COCOA specification and to implement only the methods for compensation within the VML schema.

5.4 Implementation of the COACT Model within the TRANSCOOP Run-Time System

This section describes the implementation of the COACT cooperative transaction model as part of the TRANSCOOP runtime system. The runtime environment forms an application-independent nucleus to build cooperative applications on top of it and is implemented as an extension of the object-oriented DBMS VODAK, i.e., all cooperation facilities are supported

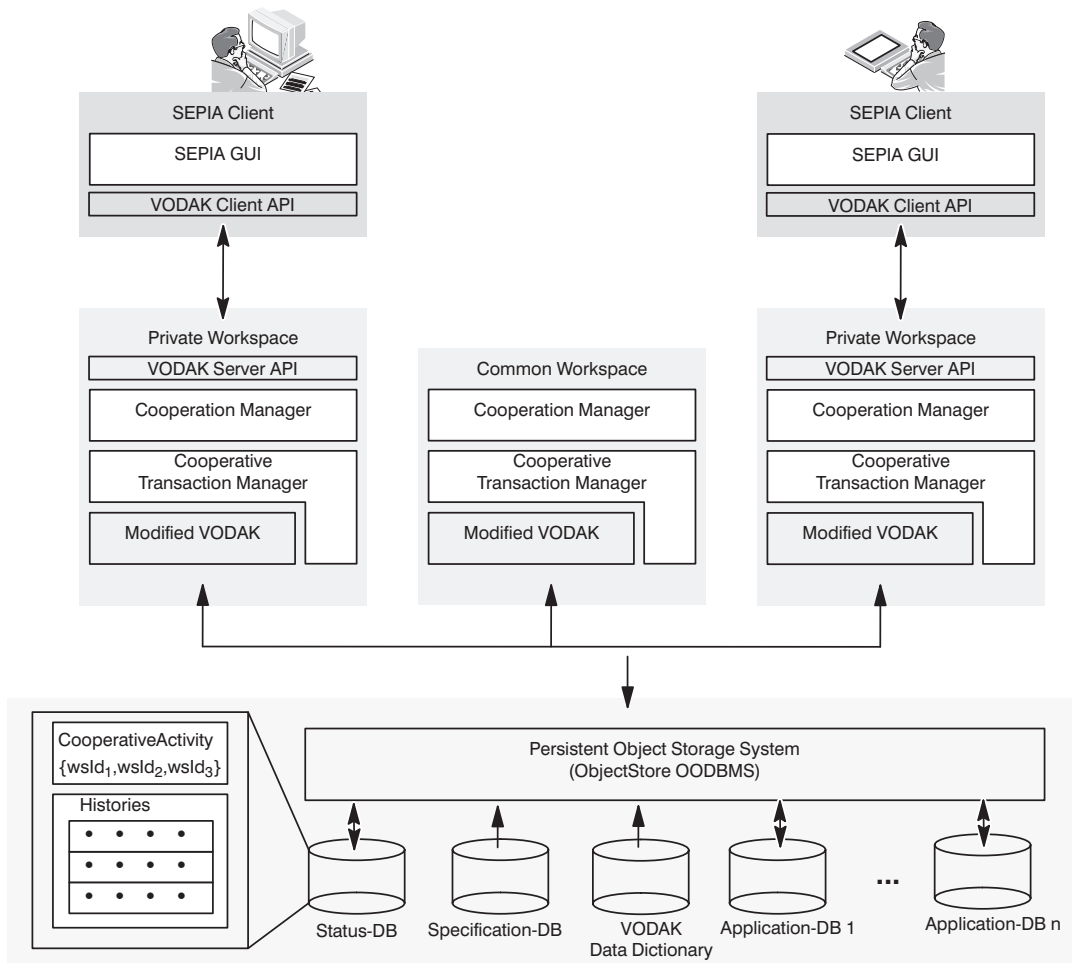


Figure 5.8: The Runtime Architecture of the TRANSCOOP Demonstrator System.

as DBMS services [WEF⁺96, dBLP⁺96]. The overall architecture of the TRANSCOOP runtime system is shown in Figure 5.8.

The design of the runtime environment required an extension of the centralized VODAK architecture in order to meet the architectural and conceptual requirements posed by the COACT cooperative transaction model. Each workspace is realized by means of a separate application database as well as a separate operating system process consisting of a modified VODAK system, the cooperative transaction manager (CTM), containing the merge functionality, and the cooperation manager (CM) governing the execution of steps as specified in a CoCoA specification. The status database and the specification database are implemented as additional ObjectStore databases which are shared among all workspaces. On top of each private workspace, an extended SEPIA graphical user interface is located.

Users can join and leave an ongoing cooperative activity before it is finished. If a user joins again, the user can decide to start with the initial state of the cooperative activity or to

continue with the previous state of the private workspace (as it was at the time the user last left the cooperative activity). The latter possibility corresponds to a suspend and resume facility. The termination of the cooperative activity is possible if a termination state is reached which is evaluated within the step mechanism of CoCoA. On termination, all private application databases are removed and the application database of the common workspace represents the final result.

The cooperative transaction manager offers functions to preview the progress of co-workers by means of querying their histories in the status database, and provides operations for both the calculation of possible merge alternatives and the enforcement of a particular exchange by merging workspace histories according to the users selection.

Most of the COACT-specific functionality of the runtime system is realized on top of VODAK, e.g., within the CTM or CM. This makes the implementation less dependent on the VODAK system. In addition, it reduces the complexity of the system as the interface to existing components can be small. On the other hand, functionality that is closely related to the existing components of VODAK is realized as extensions of existing VODAK modules. This is the case for the management of the different workspaces and their object versions.

Some of the design decisions taken in the VODAK architecture turned out to be quite favorable for the implementation of the COACT model. This holds for the logical object identification, the object management, the dynamic method handling, and the extensible data dictionary. In the remainder of this section, we describe the different components in more detail. We focus on the cooperative transaction manager, since it implements most of the functionality of the COACT model, which is the main contribution of this thesis.

5.4.1 Realization of the Workspace Concept

The desired workspace functionality of COACT requires the maintenance of multiple private versions of a data object, instead of a single shared version. Rather than implementing a special purpose version management strategy, we decided to revise the memory architecture of VODAK.

In the original VODAK system, all users share the same object buffer, thus, sharing all data objects contained in an application database. In the TRANSCOOP system, we assign to each workspace a non-shared object buffer containing the private object versions of the respective user. Since the object buffer is then private, we can bypass the VODAK transaction manager and fully rely on the ObjectStore transaction management.

The replication of the VODAK system (including its private object buffer) as well as the application database makes sure that the data objects of different workspaces are isolated. The application database is replicated at the start of a cooperative activity and an additional copy is generated when a new user joins the cooperative activity. Moreover, a new workspace, i.e., operating system process, is started for this user which manages the newly created application database. How object identifiers are handled is described on page 181 in Section 5.4.4.

When a user wants to access a data object, it is loaded from the private application database

to the private object buffer that belongs to this workspace. Modifications are written back only to the private application database associated with the workspace. The different workspaces can be located on the same machine or be distributed among different machines. At the end of a cooperative activity all private application databases are removed and the application database of the common workspace represents the final result.

5.4.2 The Status Database

The history merging algorithm we have implemented uses the histories of different workspaces at the same time. To provide efficient access to *all* workspace histories of an ongoing cooperative activity, we decided to store all the workspace histories at a central place, i.e., the status database.

Moreover, the status database contains the runtime structure of the COCoA step specification and provides general administrative information among different workspaces such as the current participants of the cooperative activity or the operations enabled by the current step.

The workspace histories are realized as double-linked lists within the central status database. When a data operation is invoked in a workspace, the corresponding history entry is first written by the CTM to the status database (write-ahead logging). Since the status database resides within the ObjectStore DBMS, we can make use of the ObjectStore transaction management for multi-user access and recovery of runtime data structures.

This design requires permanent availability of the central status database, but in case of a failure within the workspace, it allows forward recovery of the workspaces by re-executing the logged operations as well as forward recovery of the status of the cooperative activity.

However, for different application domains like mobile environments, which are outside the scope of the prototype implementation, other realizations are more appropriate. For example, local storage of workspace histories would increase the independence of the central storage system [KTW97b, KTW97a]. In the implementation of the cooperative calendar manager [Kar98] based on the COACT model the histories are stored locally at each workspace.

5.4.3 Cooperation Manager

The cooperation manager (CM) is the module of the runtime system which makes sure that the work proceeds in line with the organizational constraints of the COCoA specification. This means that the CM has to maintain the organizational step structure and to check whether the requested operations are allowed by the currently active steps. To perform this task, the CM is located in the architecture of the TRANSCOOP runtime system between the extended VODAK server API and the CTM (cf. Figure 5.9). All user requests pass the CM before they can be executed.

Possible operations that have to be checked by the CM include the ordinary data operations, exchange meta-operations, and communications initiating step transitions. Every time one of these operations is requested, the CM checks whether the requested operation is enabled. This

check depends on the parameters of the operation, on the role of the user who wants to execute it, and on the current state of the cooperative activity.

Data operations and exchange meta-operations are passed to the CTM if they are enabled (cf. Figure 5.9), communications are processed within the CM itself (cf. Figure 5.9). Communications change the internal state of the CM (persistently stored in the status database) as they trigger a step transition. Operations which are not enabled by one of the active steps are rejected by the CM.

In addition to checking operations when they are requested by the user, the CM also provides facilities to inform the user about the currently active steps and the enabled operations. This gives the user the possibility of preplanning his future activities.

The CM makes use of both the specification database and the status database. The specification database is used to store the organizational part of a COCOA specification in an internal format. The status database holds the dynamic runtime data structures.

For more information on the implementation of the cooperation manager, we refer the reader to [WEF⁺96, dBLP⁺96].

5.4.4 Cooperative Transaction Manager

The cooperative transaction manager (CTM) is the module that implements the history merging functionality of COACT and forms the core component of the demonstrator system.

A major task of the CTM is to generate and maintain the workspace histories. To make sure that each data operation executed in a workspace is recorded in the workspace history, all requests to execute an operation that are sent to the VODAK message handler are intercepted by the CTM (cf. Figure 5.9). The CTM uses the information contained in the operation call to generate an entry in the workspace history, which is persistently stored in the status database. Once a history entry is generated it will never be removed from the history. We made this implementation decision to allow for further extensions to analyze the working process after the end of a cooperative activity. If a history entry has to be logically removed from the history, i.e., it is compensated, both the compensated history entry and its compensating operation are marked as invalid in the history. This implies that they are ignored by the CTM for future checks for dependencies or conflicts.

To initiate an exchange meta-operation, the CTM provides the possibility of querying the workspace histories of all participants, e.g., his own workspace history and the history of a co-worker in case of an *import* exchange meta-operation. A user can request the whole workspace history or restrict the search to operations of certain types or specific receiver objects. The CTM offers the function `request_work` for this purpose.

To perform a merge, the CTM offers two functions. The first function `check_merge` allows the user to check for possible merge conflicts. By means of the graphical user interface, the user selects a set of operations to be merged from a source history into a destination history. The function `check_merge` then returns those operations of the destination history which need to be compensated to integrate the selected operations. The operations to be compensated

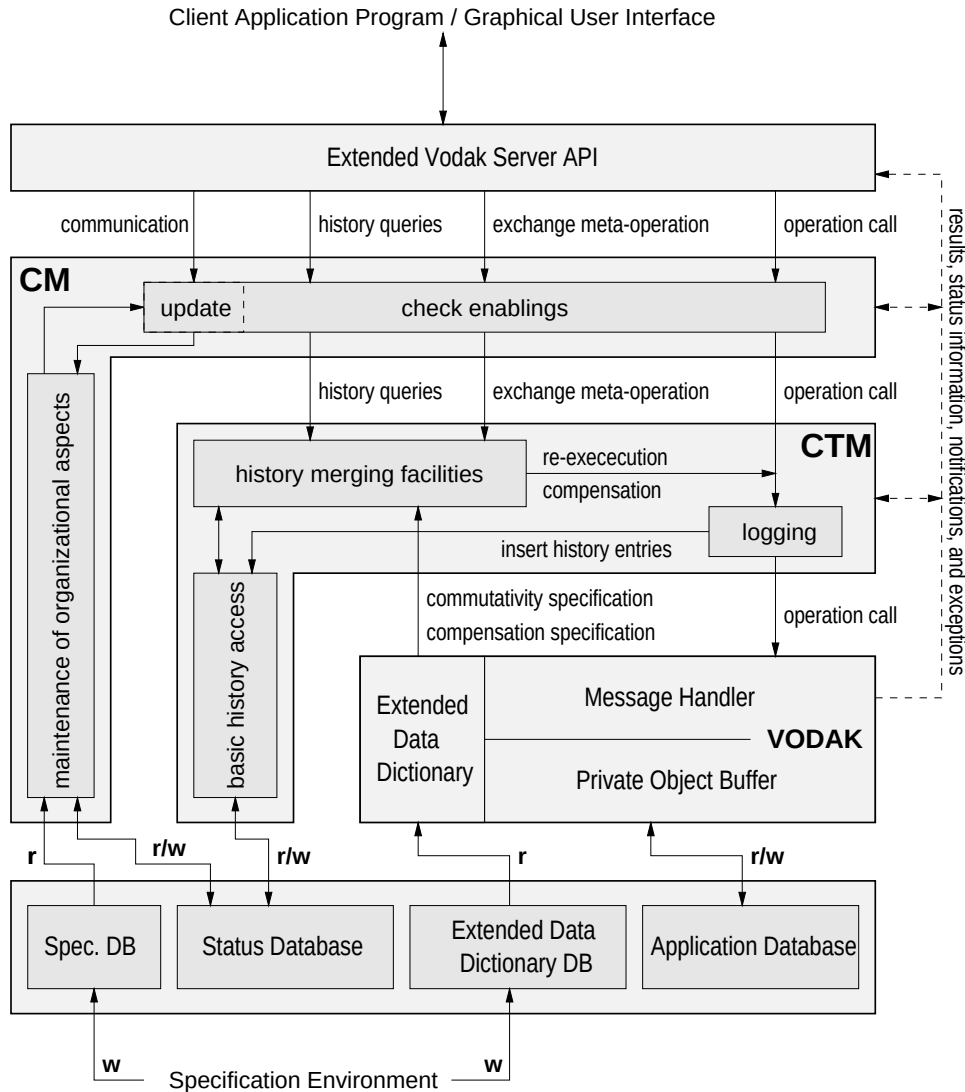


Figure 5.9: Interaction of the Different Components in the Runtime System.

are grouped by the operations selected by the user. Hence, the user can see what implications the merge of each operation has (for details on the visualization, see Section 5.4.5). The function `check_merge` detects not only direct conflicts, i.e., operations in the destination history which do not commute forward with one of the selected operations, but also indirect conflicts. For example, for each selected operation the minimal closed subhistory is calculated and all operations in the destination history transitively conflicting with one of these operations are detected (minimal units of compensation).

To perform the actual integration task the CTM provides the function `force_merge`. This function has similar input as `check_merge`. It compensates and re-executes operations in the destination workspace, as necessary, using the commutativity specification and compensation function stored in the extended VODAK data dictionary (cf. Figure 5.9). Merge conflicts

are resolved by compensating all conflicting operations and their dependent operations in the destination history.

To illustrate the use of these CTM functions, let us look at the typical sequence of function calls to the CTM to perform an *import*. The user first browses the workspace histories of other users (or the common workspace) by means of calls to the CTM function `request_work`. After choosing certain operations of a co-worker (or the common workspace), the user calls `check_merge` to get information about work which might be discarded in his workspace due to this merge. The information obtained this way might lead to the decision not to import certain operations. If the user has finally decided to import a certain set of operations which either have no merge conflicts or only those which the user is willing to tolerate, the user calls `force_merge` to complete the merge. The realization of the *save* exchange meta-operation is similar to *import*, with the difference that the saving user performs the merge in the common workspace, instead of in his private workspace.

In contrast to *import* and *save*, in a *delegate* exchange meta-operations the user who initiates the exchange meta-operation is different from the user who integrates the delegated data operations into the destination workspace. Therefore, the delegation operation is executed in two steps. In the first step, the delegator selects the operations to be exchanged in the history of his own workspace. This exchange history is then stored in a dedicated location of the status database. After that, the delegatee is notified of this delegation. From there the delegatee can retrieve, on demand, the exchange history and integrate it into his private workspace similar to the case of *import*.

Another important functionality offered by the CTM is the selective undo facility. The function `check_undo` computes the minimal unit of compensation for a set of selected operations. The function `force_undo` executes the compensation sequence for the computed minimal unit of compensation.

Special care has to be taken for the re-execution of operations that create new data objects. Remember that in the COACT model the information exchange is performed by exchanging operations instead of data objects. Therefore, newly created data objects are made available to co-workers by exchanging the operation that creates the objects. When an object is created in VODAK, it is assigned a unique object identifier (OID). This OID can be viewed as a system generated input parameter to the operation. To make sure that the re-execution of the creating operation behaves analogously to the original operation execution and that all subsequent operations refer to the same object, we have to assign the same OID during re-execution. In VODAK, there is a counter which denotes the next OID that will be generated. To enable the correct OID generation for the re-execution, we store the value of this counter prior to the initial execution of the operation in the corresponding history entry. For the re-execution, we set the counter used for OID generation temporarily to the stored value. Since the merge algorithm guarantees that the re-execution behaves like the original execution, we know that the same objects will be generated in the same order. Hence, at the time of each OID generation the counter has the same value as that before the initial execution. This ensures that the OIDs generated

during re-execution are the same as those during the initial execution of the operation. After the re-execution of the operation, the counter is reset to its original value.

For more information on the implementation of the cooperative transaction manager and its full set of functions offered, we refer the reader to [KTW96a, Sal96, WEF⁺96, dBLP⁺96].

5.4.5 Extended Graphical User Interface

The graphical user interface of SEPIA (cf. Section 5.3.3) has been extended to make the functionality provided by the CM and the CTM available to users. Because user interface design was beyond the scope of the TRANSCOOP project, work on the extended graphical user interface (EGUI) has been minimal. We have only implemented the functionality that is necessary to demonstrate the applicability and usability of the TRANSCOOP concepts, in particular the COACT cooperative transaction model.

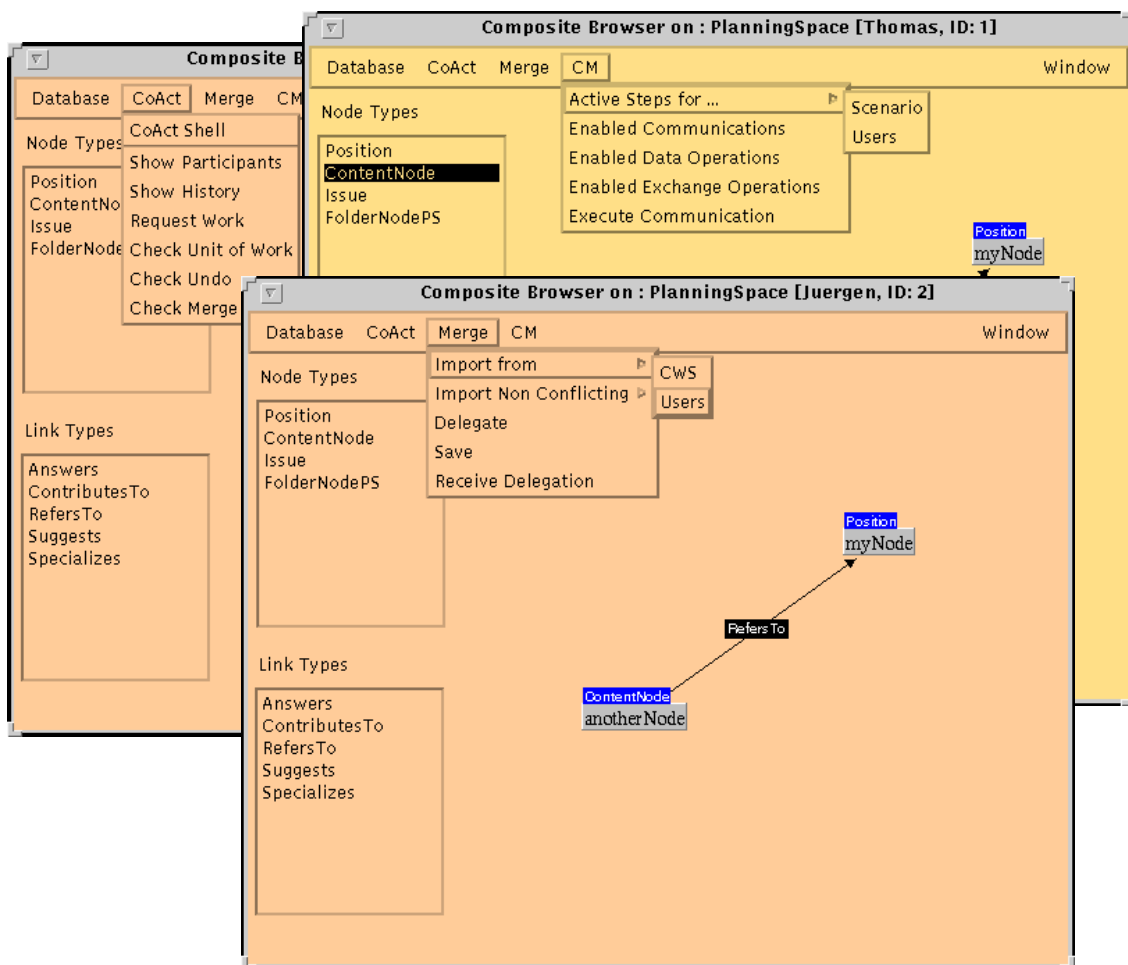


Figure 5.10: Pull-Down Menus for Cooperation Functionality.

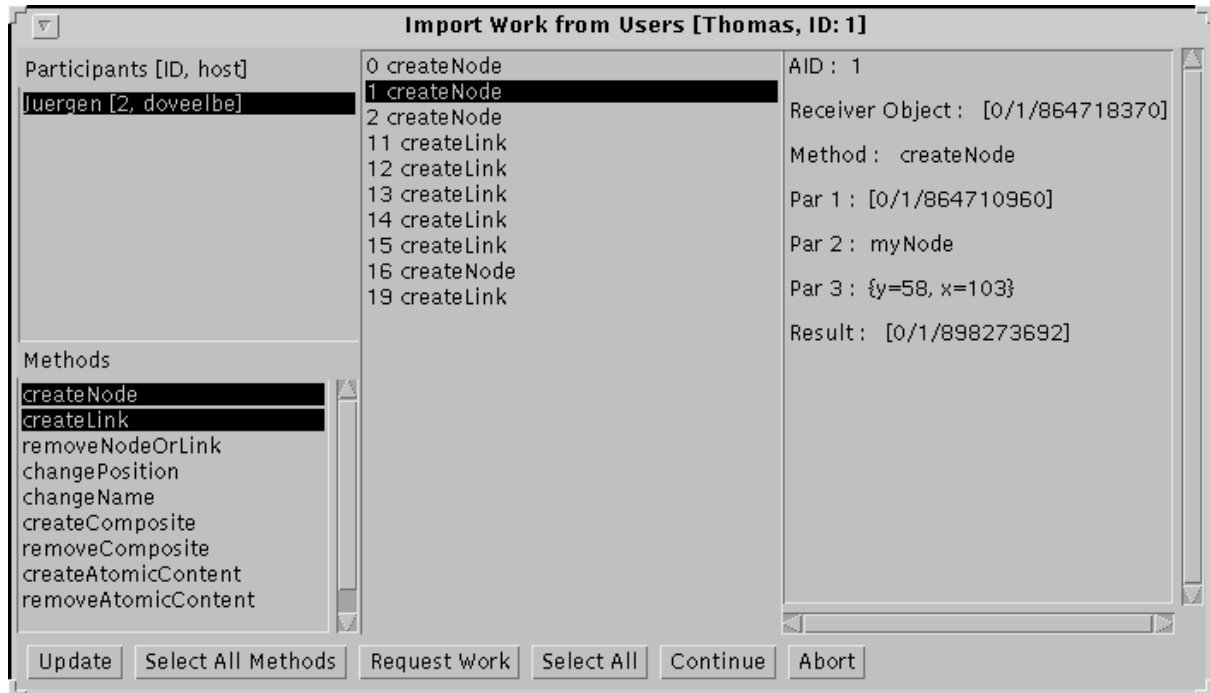


Figure 5.11: Selecting Operations for Import.

The functionality of the EGUI can be accessed by different pull-down menus which have been added to the SEPIA user interface (cf. Figure 5.10). One menu (titled “Merge” in Figure 5.10) allows to execute exchange meta-operations. It has menu entries to initiate an *import*, *delegate*, or *save* exchange meta-operation as well as to receive a delegation. Another pull-down menu (titled “COAct” in Figure 5.10) allows to query workspace histories, to check for consistent units of work and potential merge conflicts, and to selectively undo operations in the own private workspace (via the “Check Undo” entry). A third pull-down menu (titled “CM” in Figure 5.10) is used to query the organizational status of the cooperative activity and to initiate step transitions (see below).

In the following, we explain how to import information into a private workspace in more detail. When initiating an *import* meta-operation, the user can first decide by means of a sub-menu whether to import from the workspace of a co-worker or from the common workspace. After selecting one option, a window with three “columns” is presented (cf. Figure 5.11).

If the private workspace of a co-worker has been selected as the source of the merge, all co-workers that are actually active in the cooperative activity are listed in the frame titled “Participants”. The user can select a workspace by clicking on the entry for a co-worker. In case of a merge from the common workspace, this choice is not necessary, as there is only one unique common workspace per cooperative activity. In the list box titled “Methods” all invocable operation types are listed. By selecting only some of these operation types, the

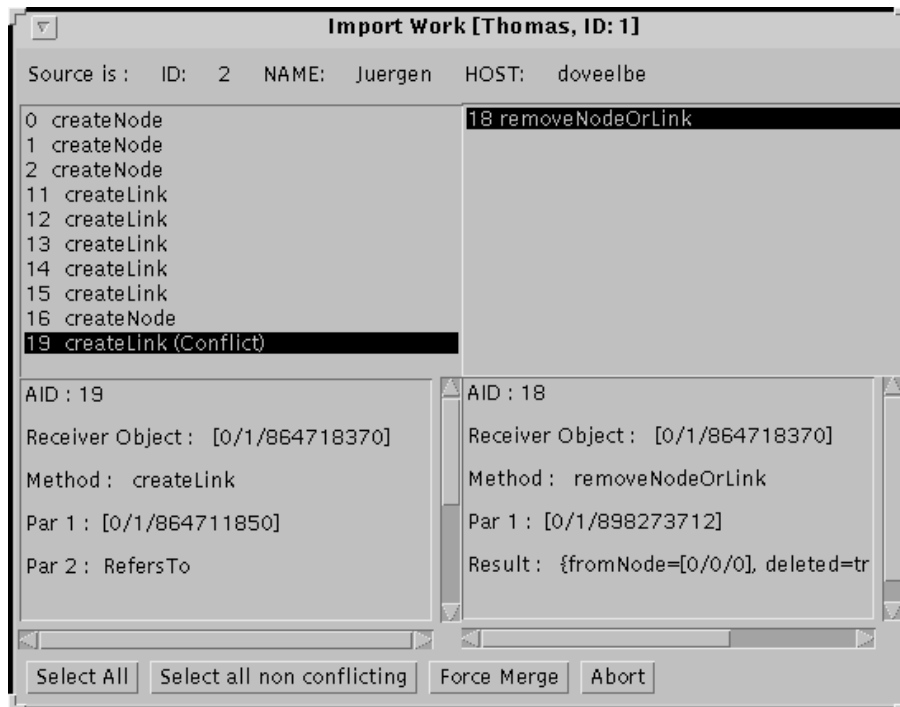


Figure 5.12: Analyzing Merge Conflicts.

user can constrain the subhistory to inspect. The button “Select All Methods” selects the whole history. By pressing the button “Request Work”, the selected subhistory of the source workspace is visualized in the list in the middle of the window. Clicking on an operation gives more details on this operation in the right column of the window¹. Using the mouse, the user can now select the operations to be imported. Pressing the button “Select All” selects the whole subhistory for the import.

If the user has made a choice, he can press the button “Continue”. This activates the function `check_merge` of the CTM and opens another window (cf. Figure 5.12). The upper left part of the window shows the selected history to be imported. Operations which have conflicting operations in the destination history are marked by the tag “(Conflict)” in the list. Clicking on an operation shows the conflicting operations in the upper right part of the window. The lower part of the window is used to provide details of selected operations. Having seen the additional information about merge conflicts, the user can decide which operations to import. The merge is completed when the user clicks on the button “Force Merge” that invokes the CTM function `force_merge`. Pressing the “Abort” button aborts the merge process.

Besides the exchange meta-operations, the EGUI provides additional functionality related to the CTM that can be accessed via the “COACT” pull-down menu (cf. Figure 5.10). On the

¹For historical reasons, the unique operation identifier is called action identifier (AID) in the EGUI.

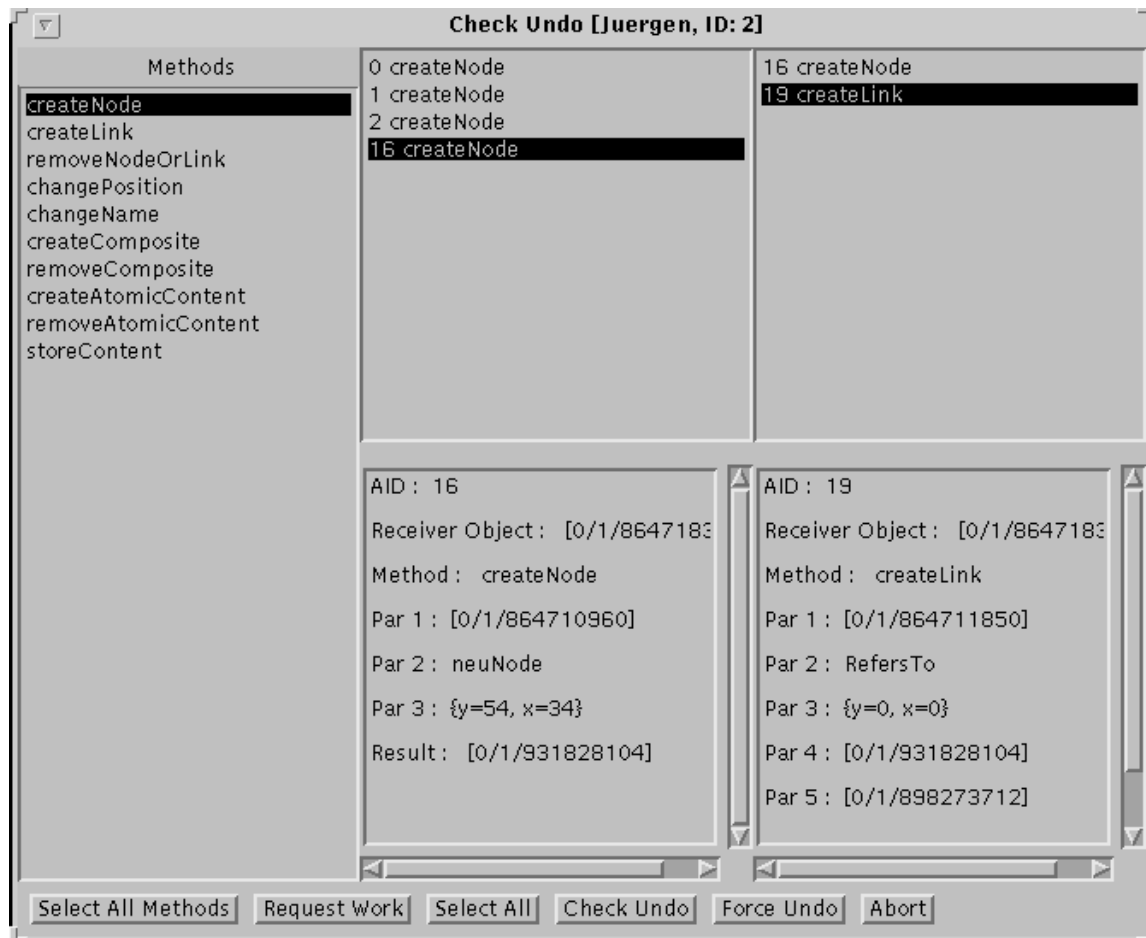


Figure 5.13: Undoing Operations.

one hand, the user can access functions of the CTM which perform certain subtasks of the merge process like showing the list of co-workers, browsing workspace histories, calculating consistent units of work, or checking for merge conflicts. This can be used for debugging purposes and to experiment with the system.

On the other hand, the user has access to additional functionality. The most important one is to retract decisions by undoing parts of his own work. When the user selects operations to be compensated (cf. Figure 5.13), he is informed about those operations that depend on the operations to be compensated, i.e., the minimal unit of compensation (cf. Figure 5.13). If the user confirms the undo, the dependent operations will be undone, too.

A separate pull-down menu is used to interact with the CM. The menu entries allow to get information about the currently active steps, the enabled operations, i.e., data operations, exchange meta-operations, and communications, and to initiate a communication. In the visualization of the data operations the user is informed about all enabled operation types and, for each type, about possible restrictions on parameters. For the exchange meta-operations, all

workspaces that can be used for exchange are listed together with the enabled types of operations on each workspace, i.e., import, delegate and save. The enabled communications are visualized in the context of the steps to which they belong.

For detailed information on the implementation of the extended graphical user interface, we refer the reader to [Sar98, WEF⁺96].

5.5 Comparison with the Original SEPIA System

In this chapter, we described the TRANSCOOP demonstrator system and, in particular, the implementation of the COACT model as extension of the object-oriented DBMS VODAK. Using this system, we implemented a demonstrator application that is based on the SEPIA hypermedia authoring system. In the remainder of this chapter, we discuss how the interaction of multiple users is handled in the original implementation of a SEPIA system [SHH⁺92, HW92] and relate this to the concepts available in our SEPIA demonstrator that is based on the COACT model.

The original cooperative SEPIA system is based on a client/server-like architecture. The server is implemented in SmallTalk-80 and comprises a shared hypermedia object base implemented using the binary object storage format of SmallTalk. The client is implemented in SmallTalk-80, too, and gives users access to the shared object base via its graphical user interface. A timestamp is associated with each data object that describes the point in time when the data object was last modified. Objects modified at a client are immediately “checked-in” into the shared hypermedia object base. If somebody else has modified these objects in the meantime, the check-in is aborted. This is detected by the server using the timestamps of the data objects. If there are no conflicting modifications, the modified objects are incorporated into the shared object base and broadcasted to the other SEPIA clients, where they replace the corresponding data objects.

The main difference between the two approaches is that the original SEPIA system does not support the concept of different private workspaces—it maintains only a single global state, i.e., the shared hypermedia object base. Hence, in SEPIA each successful modification to the shared object base must be immediately multicasted to all other co-workers.

In COACT, this corresponds to a behavior where each operation that modifies an object is delegated immediately to all other co-workers. Therefore, the histories in all workspaces would be equivalent if the same conflict resolution procedure were applied in each workspace. However, in contrast to COACT, users of the original SEPIA system do not have the possibility of performing individual work and to develop individual ideas without co-worker interference. All modifications by other users working in the same activity space are visible immediately, not on demand as in COACT.

In addition to this so-called “loosely coupled mode” in SEPIA, where only the content of the activity space is updated according to the operations executed by co-workers, the users in SEPIA can also decide to switch to the “tightly coupled mode”, during which scrolling and

resizing events of the display are immediately broadcasted. Such a shared window mode is orthogonal to the COACT concepts. Therefore, it is not available in our demonstrator application.

Another issue is the treatment of dependencies and conflicts between operations. If, for example, a user reads the text in an atomic hypermedia node and later writes back the modified text, it has to be ensured that in-between these events, the text has not been changed by another user. SEPIA handles this by implicitly grouping read and write operations on the same object. In case of a timestamp conflict, only one of the users is able to execute the write operation (the user that first writes the object back). When the other user wants to write back his modifications, the user is informed that it is too late, and the work is inevitably lost.

In the COACT model, dependencies are recognized by commutativity relations. This allows a more precise detection of dependencies. The same holds for conflicts between operations of different users. They are reduced as the use of operations semantics results in fewer conflicting operations, whereas the timestamp mechanism in the original SEPIA system considers only write/write conflicts on data objects. In addition to this, conflicts that occur need not be resolved immediately, because each user works in a private workspace. The users can decide on demand, e.g., when they see the work in a broader context, which result should be in their workspace.

The history management facilities of the COACT-based SEPIA demonstrator provide additional functionality, opposed to the original SEPIA implementation. In COACT, all operations executed by co-workers are logged in the workspace histories. This allows to query the progress of the co-workers and to offer a selective undo mechanism to the users of the system. Moreover, the step mechanism implemented by the cooperation manager, together with COCOA's user roles and enabling mechanism, provide means for the coordination of the work of multiple co-workers.

Last, since COACT is implemented on top of a full-fledged DBMS, we can rely on database functionality like a declarative query language and crash recovery that are not available in the original implementation of the SEPIA hypermedia authoring system.

Chapter 6

Conclusions and Further Work

We conclude this work by presenting a short summary of the achievements of this thesis, discussing potential application areas of history merging, and describing open issues and directions of further research.

6.1 Achievements of the Dissertation

The main contribution of this dissertation is the development of a semantics-based cooperative transaction model, called COACT. The theory and techniques developed for the COACT model were preceded by a thorough analysis of requirements derived from diverse classes of cooperative applications, ranging from cooperative authoring to workflow applications. The COACT model focuses on their key requirements, i.e., interactive user control and retraction of decisions, long duration activities and relaxed atomicity, and support for multi-user cooperation on persistent data. The latter calls for the coexistence of private and shared data and for the semantically correct exchange of both final and intermediate results among co-workers in a cooperative activity.

The COACT model addresses these requirements by combining the use of private workspaces with a new synchronization mechanism, called history merging. The workspace concept allows individual participants to work independently without interfering with others or disrupting co-workers. The history merging mechanism enables co-workers to combine their work in order to form a coherent whole. To provide this flexibility, we take an operation-oriented view in COACT. The exchange of information among workspaces is accomplished by exchanging operations, instead of the data objects themselves. The consistency of shared results is determined on the basis of the type-specific semantics of the operations.

The thesis contains a comprehensive discussion of different classes of commutativity relations, together with important properties of these relations. Based on commutativity, we introduced a new and original notion of correctness of concurrent executions using the history merging paradigm. In particular, we guarantee in COACT that all workspace histories remain legal, although information is exchanged. Legality is a novel formal concept beyond conflict

serializability. This approach relaxes the isolation property of traditional transactions in a controlled way and enables cooperation among users—at the same time it ensures consistency of the workspaces.

Based on this notion of correctness, we developed basic algorithms for implementing history merging. These algorithms are characterized by the ability to dynamically identify consistent units of work, to identify relevant conflicts, and to minimize the amount of work that has to be undone in case of a conflict. Hence, the atomicity property of traditional transactions is relaxed in a controlled way. Our approach to selective compensation can also be employed as a selective undo mechanism for users, which is important for the interactive execution of cooperative activities.

COACT's conceptual model in combination with history merging is well-suited to provide the basis for the management of consistency among disconnected or mobile users who operate independently and yet, must occasionally reconcile their work with each other or with the stationary system. We have shown in Chapter 4 how to extend COACT's system architecture in order to adapt it to the special characteristics of mobile environments.

The practical part of this thesis consists of the prototypical implementation of the COACT cooperative transaction model. We have shown how the new cooperative transaction model can be mapped into a cooperative transaction manager component for an object-oriented DBMS. Upon this platform, we have re-implemented the SEPIA hypermedia authoring system as a proof-of-concept demonstrator—extending it with the new cooperation facilities provided by the COACT cooperative transaction model.

6.2 Other Application Areas of History Merging

We believe that history merging is a general synchronization mechanism that can be utilized in application contexts other than asynchronous cooperation. History merging provides directions for solving open research problems in related fields, like semi-automatic merging in versioning systems, synchronization in synchronous cooperative applications, and (re-)integration support in mobile environments and for asynchronous database replication.

Merging in Versioning Systems. One major problem in versioning systems and check-out/check-in models is the merging of different versions of data objects into a new, combined version [CW98, Kat90] (see also Section 2.3.4). Merging is a difficult process, certainly when many interrelated data objects exist. Most of the existing merge techniques are state-based, operate only on text files, and use application-specific policies. This limits their applicability in the general case [CW98].

History merging proposes a more general technique for merging versions, namely *operation-based merging* [LvO92, CW98]. Each materialized version keeps a history of operations, that models the “delta” to its successor versions. These operation sequences, together with their type-specific semantics, can then be used to guide the process of merging arbitrary versions in

the version graph at a fine-grained level. The history merge algorithm combines them into a single history, thereby proposing different alternatives for a consistent combined version, from which the user can choose. Using our selective compensation approach, it is also possible to go back and to re-construct “intermediate” versions, which can serve as the basis to construct a new branch in the version graph (like in the Timewarp system [EM97]).

A similar approach to version merging was proposed in [LvO92], but this work suffers from its vague definition of commutativity and the need to re-construct the new version beginning from the common baseline, since no compensation operations are used.

It would be interesting to evaluate history merging for version merging in the context of the versioned SEPIA hypermedia authoring system, described in [Haa96]. There, the problem of merging different versions of hypermedia documents was not addressed (some ideas about state-based merging for SEPIA hyperdocuments have been outlined in [HHH95]). Since our demonstrator system is based on (a non-versioned variant of) the SEPIA system, this implementation seems to be straightforward.

Synchronization in Synchronous Cooperative Applications. Although the COACT model focuses on asynchronous cooperation, its history merge mechanism can be utilized to construct synchronization mechanisms for synchronous cooperative applications. These systems are based on the WYSIWIS paradigm and usually have a fully replicated architecture (realtime groupware systems, cf. Section 2.3.8.1)

An example of such a system is the cooperative meeting room environment DOLPHIN [SGHH94], developed at GMD-IPSI. In DOLPHIN, each site replicates all data objects; there exists no central component. This corresponds to a variant of COACT where only private workspaces exist. If a sequence of operations is executed at one site by a user, it is immediately multicasted to *all* other sites. These operation sequences are then re-executed there; conflicts are immediately resolved. DOLPHIN uses a distributed optimistic concurrency algorithm [Sch95, SKSH96], which is based on ORESTE [KBL93]. This algorithm suffers from frequent undos and redos, caused by operations that are received out of order. Since no selective compensation is applied, this may become expensive. Moreover, the algorithm is vulnerable to network partitions or high-latency network connections.

Using history merging for synchronization in synchronous cooperative applications can provide more flexibility. For instance, only the order of backward conflicting operations have to be preserved. In case of conflicts among operations generated at different sites, history merging tries to minimize the number of operations that need to be undone. Since history merging was designed for asynchronous cooperation, it can tolerate failure situations better and can be used to realize different “coupling modes” (e.g., immediate multicast, periodic multicast, multicast on demand).

One issue we have to address for synchronous cooperative applications is how to enforce that all sites converge to equivalent states in case of quiescence. One possibility is to employ the same (pre-defined) conflict resolution policy at each site, e.g., by associating priorities to

the sites and using timestamps for operations. Further optimization possibilities, like synchronization points (cf. Section 3.5.4.3), can be used to adapt the history merge algorithm to the special characteristics of synchronous groupware systems.

Re-integration Support in Mobile Computing and Asynchronous Replication. In Section 4, we have described how to extend COACT's underlying system architecture in order to support cooperation among users in mobile environments with frequent disconnections.

However, history merging seems also to be applicable for general (ACID) transaction processing in mobile environments. During integration of mobile transactions into the stationary system, it can reduce the number of conflicts, and, thus, the number of transactions that have to be undone. As noted in [PS98], our history merging approach is a promising and well-founded basis for merging operations and transactions executed at different (mobile and stationary) sites.

This feature can also be utilized for synchronization purposes in asynchronous database replication. Asynchronous replication allows transactions to update one replica directly; the update is propagated to other replicas later on. Since asynchronous updates from different transactions can conflict, these replication conflicts have to be detected and resolved [BN97]. It seems possible to utilize techniques based on history merging to reduce the number of conflicts and manual repairs in asynchronous database replication. This is especially important for symmetric peer-to-peer replication among several database servers, which is employed, e.g., in Lotus Notes.

6.3 Further Research Topics

A number of issues are raised by this research, but could not be addressed in detail in this thesis. In the following, we discuss some of these issues and point to potential areas of future research based on this work.

Detailed Evaluation of COACT and History Merging in Different Application Contexts.

We have shown that history merging is a general synchronization mechanism that can be applied in a variety of different application domains. Although, we built two proof-of-concept demonstrator systems for the COACT model (the cooperative SEPIA system described in Section 5 and a cooperative calendar manager [Kar98]) that appeared to be successful, detailed evaluations of COACT and its history merging mechanisms remain to be done. In particular, no systematic analysis about the scalability and optimization possibilities for different kinds of cooperative applications has been performed. Also, empirical evaluations about the usability of the COACT model have not been carried out.

Additional Coordination Facilities for the COACT model.

In this thesis, we have focussed on transactional aspects of cooperative activities, i.e., data consistency in the workspaces. Coordination and organizational aspects are not directly addressed by the COACT model—in the

TRANSCOOP project they are provided by the CoCoA coordination model. Although in asynchronous work environments the need for coordination and communication is less frequent than in synchronous cooperation [EMP⁺97], it seems to be beneficial to integrate dedicated coordination mechanisms into the CoACT model.

One area of future research hereby is the support for *automatic conflict resolution* in CoACT. If a system is able to mechanically deal with conflicts, we can relieve the users from the burden of selecting manually one of the alternatives the CoACT model offers in that case. To be usable for a wide range of applications, the conflict resolution facilities must be able to specify application-specific conflict resolution policies, e.g., based on user roles and authorities, priorities, or measures about the quality and the overall benefit of different alternatives.

Additionally, the specification and enforcement of special *cooperation policies* that define when to exchange what information among whom can be of use within cooperative activities. Similar mechanisms are already available in the CoCoA coordination model, but more elaborated concepts will further enhance the usability of CoACT. This can also improve the performance of history merging, since compaction of histories is possible if synchronization points are used.

Another area of future work is the integration of *divergence control* into the CoACT model. At the moment, CoACT is rather optimistic in the sense that it allows users to work individually and postpones conflict resolution until the actual merge of two workspaces takes place. More pessimistic coordination may be useful to constrain the divergence of workspaces within a cooperative activity. For example, access control mechanisms can be employed to reserve some parts of workspaces for a period of time. Another possibility is to check for specific conflicts among workspaces periodically. If conflicts occur, the respective users can be forced to resolve them immediately. Also, enforcement of synchronization points can help to control the divergence of workspaces within a cooperative activity.

Awareness Support and User Interfaces for the CoACT model. Beyond support for cooperation and coordination, further requirements are imposed on computer system support for cooperative applications, which are not addressed by CoACT, yet. For example, group or situation awareness can be essential in cooperative activities because it gives users an indication of how to correlate and coordinate their work. To support awareness, some kind of session control and notification mechanisms are needed. Additional communication facilities, like e-mail or audio/video conferencing channels, may be required for direct negotiation of co-workers.

Special attention has to be paid in CoACT to so-called asynchronous awareness [EM97], which is concerned with what has happened in the past. Here, techniques proposed by Edwards et al. [EM97], e.g., timeline graphs and magic lenses [BSP⁺93], can be used to query, filter, visualize, and examine workspace histories, and to relate histories with each other. Providing more information about histories to the users implies that we have to record more details about the work activities of individual users, e.g., the time and the user that generated an operation.

Another issue that requires further work is *visualization and user interface support for*

merging. As user interface design was beyond the scope of this thesis, we have only implemented the functionality necessary to demonstrate the usability of the COACT cooperative transaction model. For instance, it may be more natural from the user's viewpoint to exchange partial work results in terms of data or documents, rather than in terms of operations. Mechanisms to map the exchange of data objects to the exchange of a respective set of operations have to be developed, e.g., by means of appropriate graphical user interfaces. These user interfaces should also provide advanced support for displaying, comparing, and selecting alternative merges.

Specification of Commutativity and Compensation. An important question raised by this research is the specification of cooperative activity types, in particular the specification of commutativity relations and compensation functions. In the COACT model, we assume that the designer of a cooperative activity type correctly specifies this input. However, we cannot guarantee the correctness of the specified commutativity relations and compensation functions¹. Any advanced transaction model that relies on commutativity or compensation suffers from this problem.

Specification tools that compute or prove commutativity information as well as compensation functions will improve the practical applicability of our approach and advanced transaction models in general. For example, Chrysanthis et al. describe in [CRR91] a methodology how to extract semantic information from data objects and abstract data types.

A more formal approach is followed by the University of Twente in the COMA project [SE98a, ES98, SE99], which is a follow-up to the verification efforts in TRANSCOOP. The COMA project aims at automated analysis of commutativity and compensation properties of object-oriented database schemas. The schema specifications are mapped by a schema translator to a formal model in higher-order logic. The reasoning component used is based on the higher-order logic Isabelle theorem prover [Pau94].

A first result of COMA is a software tool that assists a schema designer in reasoning about the semantic correctness of atomic operations in a database schema [SE99, SE98a]. This tool has been used to successfully prove the correctness of the compensation function of the SEPIA cooperative activity type (cf. Section 3.4.2) [ES98]. The next step is to extend the techniques to support the verification of commutativity relations as well [ES98].

¹This is similar to the assumption in traditional transaction management that the specified transactions obey the consistency property.

Bibliography

- [AA94] D. Agrawal and A. El Abbadi. A nonrestrictive concurrency protocol for object-oriented databases. *Distributed and Parallel Databases*, 2(1):7–31, 1994.
- [AAA⁺95] G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan, R. Günthör, and M. Kamath. Exotica/FMQM: Persistent message-based architecture for distributed workflow management. In *Proc. of IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*, Trondheim, Norway, August 1995.
- [AAA⁺96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the 12th IEEE International Conference on Data Engineering (ICDE)*, pages 574–581, New Orleans, Louisiana, February 1996.
- [AAAM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *IEEE Expert Journal*, 12(5), 1997.
- [AAS93] D. Agrawal, A. El Abbadi, and A. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460–486, 1993.
- [AEF⁺97] K. Aberer, S. Even, F. Faase, Kaijanranta H, J. Klingemann, A. Lehtola, O. Pihlajamaa, T. Tesch, J. Wäsch, P. Apers, and E. J. Neuhold. Transaction support for cooperative work: An overview of the TRANSCOOP project. TRANSCOOP Consortium, Document distributed at the *Workshop on Extending Data Management for Cooperative Work*, Darmstadt, Germany, June 6 1997.
- [AF95] K. Aberer and G. Fischer. Semantic query optimization for methods in object-oriented database systems. In *Proceedings of the 11th IEEE International Conference on Data Engineering (ICDE)*, pages 70–79, Taipei, Taiwan, March 1995.
- [AG95] P. Antunes and N. Guimaraes. Beyond formal processes: Augmenting workflows with group interaction techniques. In *Proceedings of the ACM Conference on Organizational Computing Systems*, pages 1–9, Milpitas, California, August 1995.
- [AG98] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley Publishing Company, 1998. Second Edition.

- [AGK⁺96] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A. El Abbadi, and C. Mohan. Exotica/FMDC: A workflow management system for mobile and disconnected clients. *Distributed and Parallel Databases*, 4(3), July 1996.
- [AK93] R. Alonso and H. Korth. Database system issues in nomadic computing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 388–392, Washington, DC, May 26–28 1993.
- [AKT⁺96] K. Aberer, J. Klingemann, T. Tesch, J. Wäsch, and E. J. Neuhold. Transaction models supporting cooperative work – The TRANSCOOP experiences. In *Proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications (CODAS '96)*, pages 467–476, Kyoto, Japan, December 5–7 1996.
- [AMY88] R. Aksyn, D. McCracken, and E. Yoderer. KMS: A distributed hypermedia systems for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, 1988.
- [ANRS92] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using flexible transactions to support multi-system telecommunication applications. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB)*, pages 65–76, Vancouver, Canada, August 1992.
- [ARSS93] P. C. Attie, M. Rusinkiewicz, A. Sheth, and M. P. Singh. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th International Conference on Very Large Databases (VLDB)*, pages 134–145, Dublin, Ireland, August 1993.
- [AS96] G. Alonso and H.-J. Scheck. Research issues in large workflow management systems. In *Proceedings of the NSF Workshop on Workflow and Process Automation for Information Systems*, Athens, Georgia, May 1996.
- [AVA⁺94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A.E. Abbadi, H.-J. Schek, and G. Weikum. Unifying concurrency control and recovery of transactions. *Information Systems*, 19(1):101–115, 1994.
- [BA94] K. Böhm and K. Aberer. An object-oriented database application for hytime document structure. In *Third International Conference on Information and Knowledge Management (CIKM)*, Gaithersburg, Maryland, December 1994.
- [Bad79] D. Z. Badal. Correctness of concurrency control and implications in distributed databases. In *Proceedings of the IEEE Computer Software and Applications Conference*, pages 588–593, Los Alamitos, California, November 1979.
- [BAN95] K. Böhm, K. Aberer, and E. J. Neuhold. Administering structured documents in digital libraries. In *Advances in Digital Libraries*, volume 916 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

- [BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1989.
- [BBdB⁺96] R. Bal, H. Balsters, R. de By, A. Bosschaart, J. Flokstra, M. van Keulen, J. Skowronek, and B. Termorshuizen. The TM Manual, version 2.0, revision f. Technical Report IMPRESS/UT-TECH-T79-001-R2, Universiteit Twente, The Netherlands, February 1996.
- [BBG⁺83] C. Beeri, P. Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha. A concurrency control theory for nested transaction. In *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 45–62, Montreal, Canada, August 1983.
- [BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.
- [BC91] K.P. Birman and R. Cooper. The ISIS project: Real experience with a fault-tolerant programming system. *ACM Operating Systems Review*, 21(2):103–107, 1991.
- [BdBZ93] H. Balsters, R. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In *Proceedings of the Seventh European Conference on Object-Oriented Programming*, pages 161–184, Kaiserslautern, Germany, 1993.
- [BDG⁺94] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and A. Ramamritham. Asset: A system for supporting extended transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 44–54, Minneapolis, Minnesota, May 1994.
- [BDS⁺93] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *ACM SIGMOD Record*, 22(3):23–30, 1993.
- [BE96] O. Bukhres and A. Elmagarmid, editors. *Object-oriented Multidatabase Systems*. Prentice-Hall, 1996.
- [BEK93] O. Bukhres, A. Elmagarmid, and E. Kühn. Implementation of the Flex transaction model. *IEEE Data Engineering Bulletin*, 16(2):28–32, June 1993.
- [BG81] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.
- [BGRS91] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, 1991.
- [BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

- [Bjo73] L.A. Bjork. Recovery scenarios for a DB/DC system. In *Proceedings of the 28th ACM Annual Conference*, pages 142–146, Atlanta, Georgia, 1973.
- [BK91] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.
- [BKK85] F. Bancilhon, W. Kim, and H. Korth. A model of CAD transactions. In *Proceedings of the 11th International Conference on Very Large Databases (VLDB)*, pages 25–33, Stockholm, Sweden, August 1985.
- [BKL96] S. Boll, W. Klas, and M. Löhr. Integrated database services for multimedia presentations. In S. M. Chung, editor, *Multimedia Information Storage and Management*. Kluwer Academic Publishers, 1996.
- [BLW96] A. Bapat, M. Löhr, and J. Wäsch. Hypermedia support for the integrated systems engineering environment MUSE. In *Proceedings of the Fifth International Conference on Data and Knowledge Systems in Manufacturing and Engineering (DKSME '96)*, pages 123–140, Phoenix, Arizona, October 24–25 1996.
- [BMR96] D. Barbara, S. Mehrotra, and M. Rusinkiewicz. INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management*, 7(1), 1996.
- [BN97] P. Bernstein and E. Newcomer. *Principles of Transaction Processing for the Systems Professional*. Morgan Kaufmann Publishers, Inc., 1997.
- [BOH⁺92] A. Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos, and F. A. Manola. A transaction model for active distributed objects systems. In Elmagarmid [Elm92], chapter 5, pages 123–158.
- [Bol95] S. Boll. Übermittlung und Steuerung von Präsentationsabläufen und Interaktionen in einer verteilten, multimedialen Datenbankumgebung. GMD-Studie 294, Gesellschaft für Mathematik und Datenverarbeitung, June 1995.
- [BP95] R. Barga and C. Pu. A practical and modular implementation of extended transaction models. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 206–217, Zurich, Switzerland, September 1995.
- [BP97] R. Barga and C. Pu. The reflexive transaction framework. In Jajodia and Kerschberg [JK97], chapter 3, pages 63–89.
- [BR92] B. R. Badrinath and K. Ramamritham. Semantic-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.
- [BS77] R. Bayer and M. Scholnick. Concurrency control of operations on B-trees. *Acta Informtica*, 9/1977:1–21, 1977.

- [BS95] U. Borghoff and J. Schlichter. *Rechnergestützte Gruppenarbeit: Eine Einführung in verteilte Anwendungen*. Springer Verlag, 1995.
- [BSP⁺93] E. Bier, M. Stone, K. Pier, W. Buxton, and T. DeRose. Toolglass and magic lenses. In *Proceedings of the 20th Annual ACM Conference on Computer Graphics*, pages 73–80, Anaheim, California, August 1993.
- [BSS91] K.P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.
- [BSW79] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(5):203–216, May 1979.
- [BSW88] C. Beeri, H.-J. Scheck, and G. Weikum. Multi-level transaction management: Theoretical art or practical need? In *Advances in Database Technology*, volume 303 of *Lecture Notes in Computer Science*. Springer Verlag, 1988.
- [Bur97] C. Burger. *Groupware: Kooperationsunterstützung für verteilte Anwendungen*. dpunkt Verlag, Heidelberg, 1997.
- [BW96] S. Boll and J. Wäsch. A Java application programming interface to a multimedia enhanced object-oriented DBMS. In *Proceedings of the First International Workshop on Persistence and Java (PJ1)*, Glasgow, Scotland, September 16–18 1996. Sun Microsystems Laboratories Technical Report, No. SMLI TR-96-58.
- [BWAH96] A. Bapat, J. Wäsch, K. Aberer, and J. Haake. HyperStorM: An extensible object-oriented hypermedia engine. In *Proceedings of the ACM Conference on Hypertext*, pages 203–214, Washington, D.C., March 16–20 1996. Also appeared in [FO98].
- [Cas81] M. A. Casanova. *The Concurrency Control Problem of Database Systems*. Lecture Notes in Computer Science. Springer Verlag, 1981.
- [Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, Inc., 1994.
- [CCPP98] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data & Knowledge Engineering*, 24(3):211–238, 1998.
- [CD95] R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *Proceedings of European Conference on Computer Supported Work*, pages 231–246, Stockholm, Sweden, October 1995.
- [CD96] Q. Chen and U. Dayal. A transactional nested process management system. In *Proceedings of the 12th IEEE International Conference on Data Engineering (ICDE)*, pages 566–573, New Orleans, Louisiana, February 1996.

- [CGS97] S. Ceri, P. Grefen, and G. Sanchez. WIDE: A distributed architecture for workflow management. In *Proceedings of RIDE—High Performance Database Management for Large Scale Applications*, Birmingham, England, April 1997.
- [CHRW98] A. Cichocki, S. Helal, M. Rusinkiewicz, and D. Woelk. *Workflow and Process Automation - Concepts and Technology*. Kluwer Academic Publishers, 1998.
- [CMG96] G. Canals, P. Molli, and C. Godart. Concurrency control for cooperating software processes. In *Proceedings of the International Workshop on Advanced Transaction Models and Architectures (held in conjunction with VLDB'96)*, Goa, India, August 31 – September 2 1996.
- [Con87] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.
- [CR90] P. K. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, Atlantic City, NJ, May 1990.
- [CR91] P. K. Chrysanthis and K. Ramamritham. A formalism for extended transaction model. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pages 103–112, Barcelona, Spain, September 1991.
- [CR92] P. K. Chrysanthis and K. Ramamritham. ACTA: The saga continues. In Elmagarmid [Elm92], chapter 10, pages 349–397.
- [CR93] P. K. Chrysanthis and K. Ramamritham. Delegation in ACTA to control sharing in extended transactions. *IEEE Data Engineering Bulletin*, 16(2):16–19, June 1993.
- [CR94] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.
- [CR98] A. Cichocki and M. Rusinkiewicz. Migrating workflows. In Dogac et al. [DÖBS98], pages 339–355.
- [CRR91] P. K. Chrysanthis, S. Raghuram, and K. Ramamritham. Extracting concurrency from objects: A methodology. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 108–117, 1991.
- [CW98] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.
- [Dav73] C. T. Davies. Recovery semantics for a DB/DC system. In *Proceedings of the 28th ACM Annual Conference*, pages 136–141, Atlanta, Georgia, 1973.

- [dBEV95] R. de By, S. Even, and P. Verkoulen. Functionally specified distributed transactions in co-operative scenarios. In *Proceedings of RIDE—Distributed Object Management*, pages 116–121, Taipei, Taiwan, March 1995.
- [dBKV98] R. de By, W. Klas, and J. Veijalainen, editors. *Transaction Management Support for Cooperative Applications*. Kluwer Academic Publishers, 1998.
- [dBLP⁺95a] R. de By, A. Lehtola, O. Pihlajamaa, J. Veijalainen, and J. Wäsch. A reference architecture for cooperative transaction processing systems. VTT Research Notes 1694, VTT – Technical Research Center of Finland, October 1995. ISBN 951-38-4849-3.
- [dBLP⁺95b] R. de By, A. Lehtola, O. Pihlajamaa, J. Veijalainen, and J. Wäsch. Specification of the TRANSCOOP reference architecture. Report DIII.1, Esprit-III LTR Project TRANSCOOP (P8012), April 1995. Also appeared as Memoranda Informatica 95-18, University of Twente, The Netherlands, April 1995.
- [dBLP⁺96] R. de By, A. Lehtola, O. Pihlajamaa, J. Veijalainen, and J. Wäsch. Specification of the TRANSCOOP demonstrator system. Report DIII.2, Esprit-III LTR Project TRANSCOOP (P8012), April 1996.
- [DDS97] W. Du, J. Davis, and M.-C. Shan. Flexible specification of workflow compensation scopes. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP)*, pages 309–316, Phoenix, Arizona, November 1997.
- [DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, Atlantic City, NJ, May 1990.
- [DHL91] U. Dayal, M. Hsu, and R. Ladin. A transaction model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pages 113–122, Barcelona, Spain, September 1991.
- [DIM95] DIMSYS. VODAK V4.0 User Manual. Technical Report, Arbeitspapiere der GMD 910, Gesellschaft für Mathematik und Datenverarbeitung, April 1995.
- [DÖBS98] A. Dogac, T. Özsu, A. Biliris, and T. Sellis, editors. *Workflow Management Systems and Interoperability*. NATO-ASI Series. Springer Verlag, 1998.
- [Dow95] A. Downing. Conflict resolution in symmetric replication. In *Proceedings of the European Oracle User Group Conference*, pages 167–175, Florence, Italy, April 1995.
- [DPS⁺94] A. Demers, K. Peterson, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, December 1994.

- [DS93] U. Dayal and M.-C. Shan. Issues in operation flow management for long-running activities. *IEEE Data Engineering Bulletin*, 16(2):41–44, June 1993.
- [DSW94] A. Deacon, H.-J. Schek, and G. Weikum. Semantics-based multilevel transaction management in federated systems. In *Proceedings of the 10th IEEE International Conference on Data Engineering (ICDE)*, pages 452–461, Houston, Texas, February 1994.
- [ED98] A. Elmagarmid and W. Du. Workflow management: State of the art versus state of the products. In Dogac et al. [DÖBS98], pages 1–17.
- [Edw97] W. Edwards. Flexible conflict detection and management in collaborative applications. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology (UIST)*, pages 139–148, Banff, Canada, October 1997.
- [EF94] S. Even and F. Faase. Merge options for LOTOS and TM. Report DIV.1, Esprit-III LTR Project TRANSCOOP (P8012), May 1994.
- [EFdB96] S. Even, F. Faase, and R. de By. Language features for cooperation in an object-oriented database environment. *International Journal of Cooperative Information Systems, Special Issue on Formal Methods*, 5(4):469–500, December 1996.
- [EFdBP96] S. Even, F. Faase, R. de By, and O. Pihlajamaa. The TRANSCOOP specification environment. Report DIV.4, Esprit-III LTR Project TRANSCOOP (P8012), 1996.
- [EG89] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 399–407, Portland, Oregon, May 1989.
- [EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.
- [EGR89] C. Ellis, S. Gibbs, and G. Rein. Design and use of a group editor. Technical Report STP-263-88, MCC, Austin, Texas, June 1989.
- [EGR91] C. Ellis, S. Gibbs, and G. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.
- [EJ91] S. Eherer and M. Jarke. Knowledge-base support for hypermedia co-authoring. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 465–470, Berlin, Germany, August 1991.
- [EKR95] C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proceedings of the ACM Conference on Organizational Computing Systems*, pages 10–21, Milpitas, California, August 1995.
- [Ell91] C. Ellis. Groupware: Overview and perspectives. In W. Brauer and D. Hernandez, editors, *Verteilte Künstliche Intelligenz und kooperatives Arbeiten*, volume 291 of *Informatik-Fachberichte*. Springer Verlag, 1991.

- [Ell92] C. Ellis. A model and algorithm for concurrent access within groupware. Technical Report CU-CS-616-92, University of Colorado at Boulder, October 1992.
- [ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, pages 507–518, Brisbane, Australia, 1990.
- [Elm86] A. Elmagarmid. A survey of distributed deadlock detection algorithms. *ACM SIGMOD Record*, 15(3):37–45, September 1986.
- [Elm92] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. ACM Press. Morgan Kaufmann Publishers, Inc., 1992.
- [ELMB92] A. Elmagarmid, Y. Leu, J. Mullen, and O. Bukhres. Introduction to advanced transaction models. In Elmagarmid [Elm92], chapter 2, pages 33–52.
- [EM97] W. Edwards and E. Mynatt. Timewarp: Techniques for autonomous collaboration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, pages 218–225, Atlanta, Georgia, March 1997.
- [EM99] A. Eisenberg and J. Melton. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record*, 28(1):131–138, March 1999.
- [EMP⁺97] W. Edwards, E. Mynatt, K. Peterson, M. Spreitzer, D. Terry, and M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology (UIST)*, pages 119–128, Banff, Canada, October 1997.
- [ES98] S. Even and D. Spelt. Verification of compensation requirements for the SEPIA cooperative authoring system. CTIT Technical Report 98-25, University of Twente, The Netherlands, October 1998.
- [EW94a] C. Ellis and J. Wainer. A conceptual model of groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 79–88, Chapel Hill, NC, October 1994.
- [EW94b] C. Ellis and J. Wainer. Goal-based models of collaboration. *Collaborative Computing*, 1:211–238, 1994.
- [FB96] J. Flokstra and R. Boon. The TM abstract machine (TAM). Internal working document, University of Twente, Enschede, The Netherlands, February 1996.
- [FEdB96] F. Faase, S. Even, and R. de By. An introduction to CoCoA. Report DVI.3, Esprit-III LTR Project TRANSCOOP (P8012), 1996.
- [FEdB98] F. Faase, S. Even, and R. de By. The TRANSCOOP specification environment. In de By et al. [dBKV98], chapter 6, pages 115–148.

- [FEdba97] F. Faase, S. Even, R. de By, and P. Apers. Integrating organisational and transactional aspects of cooperative activities. In *Proceedings of the Sixth International Workshop on Database Programming Languages (DBPL)*, pages 226–354, Estes Park, Colorado, 1997.
- [FKL88] R. Fish, R. Kraut, and M. Leland. Quilt: A collaborative tool for cooperative writing. In *Proceedings of the Conference on Office Information Systems*, pages 30–37, Palo Alto, California, 1988.
- [FKS81] D. Fussel, Z. Kedem, and A. Silberschatz. Deadlock removal using partial roll-back in database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 65–73, Ann Arbor, Michigan, 1981.
- [FÖ89] A. Farrang and T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.
- [FO98] P. Fankhauser and M. Ockenfeld, editors. *Integrated Publication and Information Systems – 10 Years of Research and Development*. GMD – Forschungszentrum Informationstechnik GmbH, 1998.
- [FvKS94] J. Flokstra, M. van Keulen, and J. Skowronek. The IMPRESS DDT: A database design toolbox based on a formal specification language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 506, Minneapolis, Minnesota, 1994.
- [FW94a] M. Fröhlich and M. Werner. Demonstration of the interactive Graph Visualization System *daVinci*. In *Proceedings of the DIMACS Workshop on Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, Princeton, New Jersey, 1994. Springer Verlag.
- [FW94b] M. Fröhlich and M. Werner. The Graph Visualization System *daVinci*—A User Interface for Applications. Technical Report 5/94, Department of Computer Science, Universität Bremen, September 1994.
- [FZ89] M. Fernandez and S. Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Proceedings of the International Workshop on Persistent Object Systems*, pages 341–350, January 1989. Newcastle, New South Wales.
- [FZ94] G. Foreman and J. Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(6):38–47, April 1994.
- [GAP98] N. Guimaraes, P. Antunes, and A. Pereira. The integration of workflow systems and collaboration tools. In Dogac et al. [DÖBS98], pages 222–245.
- [GB91] S. Greenberg and R. Bohnet. GroupSketch: A multi-user sketchpad for geographically distributed small groups. In *Proceedings of the Conference on Graphics Interfaces*, pages 207–215, Calgary, Canada, June 1991.

- [GH94] D. Georgakopoulos and M. Hornick. A framework for enforceable specification of extended transaction models and transactional workflows. *International Journal of Intelligent and Cooperative Information Systems*, 3(3):599–617, 1994.
- [GHKM94] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the 10th IEEE International Conference on Data Engineering (ICDE)*, pages 462–473, Houston, Texas, February 1994.
- [GHM⁺93] D. Georgakopoulos, M. Hornick, F. Manola, M. Brodie, S. Heiler, F. Nayeri, and B. Hurwitz. An extended transaction environment for workflows in distributed object computing. *IEEE Data Engineering Bulletin*, 16(2):24–27, June 1993.
- [GHM96] D. Georgakopoulos, M. Hornick, and F. Manola. Customizing transaction models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):630–649, August 1996.
- [GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview on workflow management: From process modelling to workflow automation. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.
- [GKLS94] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected operation in the Thor object-oriented database system. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, December 1994.
- [GLPT75] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proceedings of the First International Conference on Very Large Databases (VLDB)*, pages 528–451, Framingham, Massachusetts, September 1975.
- [GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in distributed databases. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.
- [GM94] S. Greenberg and D. Marwood. Real-time groupware and its effect on the interface. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 207–217, Chapel Hill, NC, October 1994.
- [GMGK⁺91] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested SAGAs. *IEEE Data Engineering Bulletin*, 14(1):14–18, March 1991.
- [GMS87] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, San Fransisco, California, 1987.

- [GP96] N. Guimaraes and A. Pereira. Workflow modeling, automation, and augmentation. In *Proceedings of the NSF Workshop on Workflow and Process Automation for Information Systems*, Athens, Georgia, May 1996.
- [GP97] J. Grudin and S. Poltrock. Computer supported cooperative work and groupware. *Advances in Computers*, 45:269–320, 1997.
- [GPN91] J. Geller, Y. Perl, and E.J. Neuhold. Structural schema integration in heterogeneous multi database systems using the dual model. In *Proceedings of the First IEEE International Workshop on Interoperability in Multidatabase Systems*, pages 200–203, Kyoto, Japan, 1991.
- [GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.
- [Gra78] J. Gray. Notes on database operating systems. In R. Bayer and R. Graham, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer Verlag, 1978.
- [Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Databases (VLDB)*, pages 144–154, Cannes, France, September 1981.
- [GRL94] D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin. Chronological scheduling of transactions with temporal dependencies. *VLDB Journal*, 3(1):1–28, January 1994.
- [GS87] I. Greif and S. Sarin. Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2):187–211, April 1987.
- [GT98] D. Georgakopoulos and A. Tsalgatidou. Technology and tools for comprehensive business process lifecycle management. In Dogac et al. [DÖBS98], pages 356–395.
- [Gün93] R. Günthör. Extended transaction processing based on dependency rules. In *Proceedings of RIDE–Interoperability in Multidatabase Systems*, pages 207–214, Vienna, Austria, April 1993.
- [Gün96] R. Günthör. The dependency manager: A base service for transactional workflow management. In *Proceedings of RIDE–Interoperability of Nontraditional Database Systems*, pages 86–95, New Orleans, Louisiana, February 1996.
- [GVBA97] P. Grefen, J. Vonk, E. Boertjes, and P. Apers. Two-layer transaction management for workflow management applications. In *Proceedings of the 8th International Conference on Database and Expert Systems Applications (DEXA)*, pages 430–439, Toulouse, France, September 1997.
- [Haa95] J. Haake. *Autorensysteme für die kooperative Erstellung von Hypermedia–Dokumenten*. PhD thesis, Darmstadt University of Technology, 1995.

- [Haa96] A. Haake. *Versionenunterstützung für strukturierte Hyperdokumente im elektronischen Publizieren*. PhD thesis, Darmstadt University of Technology, 1996.
- [Had83] V. Hadzilacos. An operational method for database reliability. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 244–256, Atlanta, Georgia, March 1983.
- [Had88] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, January 1988.
- [Hal88] F.G. Halasz. Reflection on Notecards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, July 1988.
- [Här84] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 13(9):111–120, June 1984.
- [HCL90] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the Seventh ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 331–343, Nashville, Tennessee, April 1990.
- [Her86] M. Herlihy. Optimistic concurrency control for abstract data types. In *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, pages 212–217, Calgary, Canada, August 1986.
- [Her90] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.
- [Her95] T. Herrmann. Workflow management systems: Ensuring organizational flexibility by possibilities of adaption and negotiation. In *Proceedings of the ACM Conference on Organizational Computing Systems*, pages 83–94, Milpitas, California, August 1995.
- [HF80] J.R. Hayes and L. Flowers. Identifying the organization of writing processes. In *Cognitive Processes in Writing*, pages 3–30. Lawrence Erlbaum, 1980.
- [HF86] J.R. Hayes and L. Flowers. Writing research and the writer. *American Psychologist*, 41(10):1106–1113, 1986.
- [HH93] L. B. Houston and P. Honeyman. Disconnected operation for AFS. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*, Cambridge, Massachusetts, August 1993.
- [HHH95] A. Haake, J. Haake, and D. Hicks. On merging hypertext networks. In *Proceedings of the Workshop on the Role of Version Control in CSCW Applications*, Stockholm, Sweden, September 1995. Appeared as GMD-Studie 289, May 1996.

- [HHJ⁺99] P. Heint, S. Horn, S. Jablonski, J. Neeb, K. Stein, M. Teschke, and A. Ulbrich. A comprehensive approach to flexibility in workflow management systems. In *Proceedings of the International Joint Conference on Work Activities and Collaboration*, pages 79–86, San Francisco, USA, February 1999.
- [HHSW96] Y. Han, J. Himminghofer, T. Schaaf, and D. Wibarski. Management of workflow resources to support runtime adaptability and systems evaluation. In *Proceedings of the International Conference on Knowledge Engineering*, Basel, Switzerland, October 1996.
- [HKR⁺92] B. Haan, P. Kahn, V. Riley, H. Coombs, and N. Meyrowitz. IRIS hypermedia services. *Communications of the ACM*, 35(1):36–51, 1992.
- [Hon88] M. Honda. Support for parallel development in the Sun network software environment. In *Proceedings of the Second International Workshop on Computer-Aided Software Engineering*, pages 5–5 – 5–7, Cambridge, Massachusetts, July 1988.
- [HPGP92] J. S. Heidemann, T. W. Page, R. G. Guy, and G. J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*, November 1992.
- [HR83] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.
- [HR87a] T. Härder and K. Rothermel. Concepts for transaction recovery in nested transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 239–248, San Fransisco, California, December 1987.
- [HR87b] T. Härder and K. Rothermel. Concurrency control issues in nested transactions. Technical Report 95120–6099, IBM Research Laboratory, Almaden Research Center, San Jose, California, 1987.
- [HS98] Y. Han and A. Sheth. On adaptive workflow modeling. In *Proc. 4th International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, July 1998.
- [HSS96] P. Heint, H. Schuster, and K. Stein. Behandlung von Ad-hoc-Workflows im MOBILE Workflow-Modell. In *Proceedings of STAK '96 - Softwaretechnik in Automation und Kommunikation*, Munich, Germany, March 1996.
- [Hsu93] M. Hsu, editor. *Special Issue on Workflow and Extended Transaction Systems, Data Engineering Bulletin*, volume 16, number 2, June 1993.
- [HT93] J. Hannemann and M. Thüning. Schreiben als Designproblem: Kognitive Grundlagen einer Hypertext Autorenumgebung. *Kognitionswissenschaft*, Dezember 1993.

- [HW88] M. Herlihy and W. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the Seventh ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 201–210, Austin, Texas, March 1988.
- [HW92] J. Haake and B. Wilson. Supporting collaborative writing of hyperdocuments in SEPIA. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 138–146, Toronto, Canada, October 1992.
- [IA97] ISO-ANSI. Database language SQL - Part 2: Foundation (SQL/Foundation). Technical Report DBL:BBN-008 and X3H2-97-315, ISO-ANSI, October 1997. ISO-ANSI Working Draft.
- [IB93] T. Imielinski and B. Badrinath. Data management for mobile computing. *ACM SIGMOD Record*, 22(1):34–39, March 1993.
- [IB94] T. Imielinski and B. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10):18–28, October 1994.
- [IK95] T. Imielinski and H. Korth. Report from the MOBIDATA - NSF workshop on mobile and wireless information systems, May 1995.
- [IK96a] T. Imielienski and H. Korth, editors. *Mobile Computing*. Kluwer Academic Publishers, 1996.
- [IK96b] T. Imielinski and H. Korth. Introduction to mobile computing. In Imielienski and Korth [IK96a], chapter 1, pages 1–43.
- [IKG92] H. Ishii, M. Kobayashi, and J. Grudin. Integrating inter-personal and shared workspace: Clearboard design and experiments. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 33–42, Toronto, Canada, November 1992.
- [IT89] P. Irish and R. Trigg. Supporting collaboration in hypermedia. *Journal of the American Society on Information Science*, 40(3):192–199, 1989.
- [JdT⁺95] A. Joseph, A. deLespinnasse, J. Tauber, D. Gifford, and M. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 156–171, Copper Mountain Resort, Colorado, December 1995.
- [Jef85] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [JJKF89] U. Jahn, M. Jarke, K. Kreplin, and M. Farusi. CoAUTHOR: A hypermedia group authoring environment. In *Proceedings of the 1st European Conference on Computer Supported Cooperative Work*, pages 226–244, Gatwick, U.K., September 1989.

- [JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.
- [JLP⁺95] J. Juopperi, A. Lehtola, O. Pihlajamaa, A. Sladek, and J. Veijalainen. Usability of some workflow products in an inter-organizational setting. In *Proc. of IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*, Trondheim, Norway, August 1995.
- [JM86] D. Jefferson and A. Motro. The time warp mechanism for database concurrency control. In *Proceedings of the Second IEEE International Conference on Data Engineering (ICDE)*, pages 474–481, Los Angeles, California, February 1986.
- [JNRS93] W. Jin, L. Ness, M. Rusinkiewicz, and A. Sheth. Concurrency control and recovery of multidatabase work flows in telecommunication applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 456–459, Washington, D.C., June 1993.
- [JTK97] A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.
- [Kai90] G. E. Kaiser. A flexible transaction model for software engineering. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 560–567, Los Angeles, California, 1990.
- [Kai93] G. E. Kaiser. MARVEL 3.1: A multi-user software development environment. In *Proceedings of the International Symposium on Logic Programming*, Vancouver, Canada, 1993.
- [Kai95] G. E. Kaiser. Cooperative transactions for multiuser environments. In Kim [Kim95], chapter 20, pages 409–433.
- [KAN94] W. Klas, K. Aberer, and E. J. Neuhold. Object-oriented modelling for hypermedia systems using the VODAK modelling language (VML). In A. Dogac, T. Özsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, NATO-ASI Series, pages 389–443. Springer Verlag, 1994.
- [Kar98] S. Karrer. Konzeption und Implementierung eines kooperativen Terminkalenders auf Basis des Transaktionsmodells COACT. Diplomarbeit, Johann-Wolfgang-Goethe University, Frankfurt / GMD-IPSI, April 1998.
- [Kat90] R. H. Katz. Towards a unified framework for version modelling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, 1990.
- [Kat94] R. H. Katz. Adaption and mobility in wireless information systems. *IEEE Personal Communications*, pages 6–17, First Quarter 1994.
- [KB95] S. Khoshafian and M. Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing*. John Wiley & Sons, 1995.

- [KBGW91] W. Kim, N. Ballou, J. F. Garza, and D. Woelk. A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.
- [KBL93] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, pages 195–202, Pittsburgh, May 1993.
- [KdBV98] W. Klas, R. de By, and J. Veijalainen. Introduction. In de By et al. [dBKV98], chapter 1, pages 1–9.
- [KE98] J. Klingemann and S. Even. The TRANSCOOP demonstrator system. In de By et al. [dBKV98], chapter 8, pages 173–192.
- [KF87] G. E. Kaiser and P. H. Feiler. Intelligent assistance without artificial intelligence. In *Proceedings of the 32th IEEE Computer Society International Conference*, pages 236–241, San Francisco, California, USA, February 1987.
- [KFA94] W. Klas, G. Fischer, and K. Aberer. Integrating a relational database system into vodak using its metaclass concept. *Journal of Systems Integration*, 4(4), 1994.
- [KFM⁺96] W. Klas, P. Fankhauser, P. Muth, T. Rakow, and E. J. Neuhold. Database integration using the object-oriented database system VODAK. In Bukhres and Elmagarmid [BE96], pages 472–532.
- [Kim95] W. Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and beyond*. Addison-Wesley Publishing Company, 1995.
- [Kis96] J. Kistler. *Disconnected Operation in a Distributed File System*. Springer Verlag, 1996.
- [KKB87] H. Korth, W. Kim, and F. Bancilhon. On long-duration CAD transactions. *Information Systems*, 13, 1987.
- [Kle91] J. Klein. Advanced rule driven transaction management. In *Proceedings of the 36th IEEE Computer Society International Conference*, pages 562–567, San Francisco, California, USA, March 1991.
- [KLMP84] W. Kim, A. Lorie, D. McNabb, and W. Plouffe. A transaction mechanism for engineering design databases. In *Proceedings of the 10th International Conference on Very Large Databases (VLDB)*, pages 355–362, Singapore, August 1984.
- [KLS90] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, pages 95–106, Brisbane, Australia, 1990.
- [Kor83] H. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, 1983.

- [Kor95] H. Korth. The double life of the transaction abstraction: Fundamental principle and evolving system concept. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 2–6, Zurich, Switzerland, September 1995.
- [KP87] G. E. Kaiser and D. E. Perry. Workspaces and experimental databases: Automated support for software engineering. In *Proceedings of the Conference of Software Maintenance*, pages 108–114. IEEE Computer Society Press, September 1987.
- [KP90] M.J. Knister and A. Prakask. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third ACM Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, 1990.
- [KP92] G. E. Kaiser and C. Pu. Dynamic restructuring of transactions. In Elmagarmid [Elm92], chapter 8, pages 265–295.
- [KP93] M. J. Knister and A. Prakash. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems*, 6(2):135–166, 1993.
- [KPE92] E. Kühn, F. Puntigam, and A. Elmagarmid. Multidatabase transaction and query processing. In Elmagarmid [Elm92], chapter 9, pages 297–348.
- [KPS89] G. E. Kaiser, D. E. Perry, and W. M. Schell. Infuse: Fusing integration test management with change management. In *Proceedings of the 13th IEEE Computer Software and Applications Conference*, pages 552–558, Orlando, Florida, USA, September 1989.
- [KPTV95] J. Klingemann, J. Puustjärvi, T. Tesch, and J. Veijalainen. Analysis of extensions and adaptations of the open-nested transaction model. Report DV.1, Esprit-III LTR Project TRANSCOOP (P8012), 1995.
- [KR81] H. T. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.
- [KRC⁺96] P. Krychniak, M. Rusinkiewicz, A. Chichocki, A. Sheth, and G. Thomas. Bounding the effects of compensation under relaxed multi-level serializability. *Distributed and Parallel Databases*, 4(4):355–374, 1996.
- [KS88] H. Korth and G. Speegle. Formal method of correctness without serializability. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–386, Chicago, Illinois, June 1988.
- [KS90] H. Korth and G. Speegle. Long-duration transactions in software design projects. In *Proceedings of the Sixth IEEE International Conference on Data Engineering (ICDE)*, pages 568–574, Los Angeles, California, February 1990.
- [KS92] J. Kistler and M. Satyanarayanan. Disconnected operations in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

- [KS94a] H. F. Korth and G. Speegle. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, 19(3):492–535, 1994.
- [KS94b] N. Krishnakumar and A. Sheth. Specifying multi-system workflow applications in METEOR. Bellcore, Technical Memorandum, 1994.
- [KS95a] W. Klas and M. Schrefl. *Metaclasses and Their Application—Data Model Tailoring and Database Integration*, volume 943 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.
- [KS95b] N. Krishnakumar and A. Sheth. Managing heterogenous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.
- [KS95c] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file systems conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, Louisiana, January 1995.
- [KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 388–401, Austin, Texas, USA, May 1985.
- [KTW96a] J. Klingemann, T. Tesch, and J. Wäsch. Design of the TRANSCOOP cooperative transaction manager. Report DV.3, Esprit-III LTR Project TRANSCOOP (P8012), May 1996.
- [KTW96b] J. Klingemann, T. Tesch, and J. Wäsch. Semantics-based transaction management for cooperative applications. In *Proceedings of the International Workshop on Advanced Transaction Models and Architectures (held in conjunction with VLDB'96)*, pages 234–252, Goa, India, August 31 – September 2 1996.
- [KTW96c] J. Klingemann, T. Tesch, and J. Wäsch. Semantics-based transaction management for cooperative applications. Technical Report, Arbeitspapiere der GMD 990, Gesellschaft für Mathematik und Datenverarbeitung, March 1996.
- [KTW⁺96d] J. Klingemann, T. Tesch, J. Wäsch, J. Puustjärvi, and J. Veijalainen. Definition of the TRANSCOOP cooperative transaction model. Report DV.2, Esprit-III LTR Project TRANSCOOP (P8012), April 1996.
- [KTW97a] J. Klingemann, T. Tesch, and J. Wäsch. Cooperative data management and its application to mobile computing. *International Journal of Cooperative Information Systems (IJCIS)*, 6(3&4):341–365, 1997.
- [KTW97b] J. Klingemann, T. Tesch, and J. Wäsch. Enabling cooperation among disconnected mobile users. In *Proceedings of the Second International Conference on Cooperative Information Systems (CoopIS)*, pages 36–45, Kiawah Island, South Carolina, June 24–27 1997.

- [KTW97c] J. Klingemann, T. Tesch, and J. Wäsch. Enabling cooperation among disconnected mobile users. Technical Report, Arbeitspapiere der GMD 1058, Gesellschaft für Mathematik und Datenverarbeitung, March 1997.
- [KTWK98] J. Klingemann, T. Tesch, J. Wäsch, and W. Klas. The TRANSCOOP transaction model. In de By et al. [dBKV98], chapter 7, pages 149–172. Also appeared in [FO98].
- [Kum96] V. Kumar, editor. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996.
- [KW84] R. Katz and S. Weiss. Design transaction management. In *Proceedings of the 19th Design Automation Conference*, June 1984.
- [KWA98] J. Klingemann, J. Wäsch, and K. Aberer. Adaptive outsourcing in cross-organizational workflows. GMD Report 30, GMD – Forschungszentrum Informationstechnik GmbH, August 1998.
- [KWA99a] J. Klingemann, J. Wäsch, and K. Aberer. Adaptive outsourcing in cross-organizational workflows. In *Proceedings of the 11th Conference on Advanced Information Systems Engineering CAiSE*, pages 417–421, Heidelberg, Germany, June 1999. Extended version published as [KWA98].
- [KWA99b] J. Klingemann, J. Wäsch, and K. Aberer. Deriving service models in cross-organizational workflows. In *Proceedings of RIDE-Information Technology for Virtual Enterprises*, pages 100–107, Sydney, Australia, March 1999.
- [LA94] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):326–348, 1994.
- [LAA93] H. Leong, D. Agrawal, and J. Agre. Using message semantics to reduce rollback in the time warp mechanism. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, pages 309–323, Lausanne, Switzerland, September 1993.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.
- [LBL95] M. Löhr, S. Boll, and H. J. Lee. VRAPI - The VODAK Remote Application Interface. Technical Report muse-95-5, TH Darmstadt, Dept. of Computer Science, September 1995.
- [LC87] D. Leblang and R. Chase. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, 1987.
- [LdBTW98] A. Lehtola, R. de By, H. Tirri, and J. Wäsch. The TRANSCOOP architecture. In de By et al. [dBKV98], chapter 5, pages 93–113.

- [Leu91] Y. Leu. Composing multidatabase applications using flexible transactions. *IEEE Data Engineering Bulletin*, 14(1):29–33, March 1991.
- [Ley95] F. Leymann. Supporting business processes via partial backward recovery in workflow management systems. In *Proceedings of the GI-Fachtagung Datenbanksysteme in Buero, Technik und Wissenschaft*, pages 51–70, Dresden, Germany, March 1995.
- [Ley97] F. Leymann. Transaktionsunterstützung für Workflows. *Informatik Forschung und Entwicklung*, 12(2):82–90, 1997.
- [LFK88] M. Leland, R. Fish, and R. Kraut. Collaborative document production using Quilt. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 206–215, Portland, Oregon, September 1988.
- [LHW92] A. Lemke, J. Haake, and B. Wilson. WSCRAWL: The use of a shared workspace in a desktop conferencing system. Arbeitspapiere der GMD 672, Gesellschaft für Mathematik und Datenverarbeitung, August 1992.
- [Lip92] E. Lippe. *Camera: Support for Distributed Cooperative Work*. PhD thesis, Utrecht University, The Netherlands, 1992.
- [LL91] C. Lamb and G. Landis. The ObjectStore Database System. *Communications of the ACM*, 34(11):32–39, 1991.
- [LMWF94] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., 1994.
- [Lot96] Lotus Development Corporation. Lotus database management. Lotus Notes Release 4: Developer's Guide, 1996.
- [Lot97] Lotus Development Corporation. Lotus.domino. Lotus Domino Release 4.6: Developer's Guide, 1997.
- [LP97] L. Liu and C. Pu. ActivityFlow: Towards incremental specification and flexible coordination of workflow activities. In *Proceedings of the International Conference on Conceptual Modeling (ER'97)*, pages 169–182, Los Angeles, California, November 1997.
- [LP98] L. Liu and C. Pu. Methodical restructuring of complex workflow activities. In *Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE)*, pages 342–350, Orlando, Florida, USA, February 1998.
- [LR98] F. Leymann and D. Roller. Building a robust workflow management system with persistent queues and stored procedures. In *Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE)*, pages 254–258, Orlando, Florida, February 1998.

- [LS94] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *ACM Operating Systems Review*, 28(2):81–87, April 1994.
- [LvO92] E. Lippe and N. van Oosterom. Operation based merging. In *Proceedings of the Fifth ACM Symposium on Software Development Environments*, pages 78–87, Tyson's Corner, Virginia, December 1992.
- [MAGK95] C. Mohan, G. Alonso, R. Günthör, and M. Kamath. Exotica: A research perspective of workflow management systems. *IEEE Data Engineering Bulletin*, 18(1):19–26, 1995.
- [MCC98] MCC. CMI - Collaboration management infrastructure for comprehensive process and service management. Home page: www.mcc.com/projects/cmi; Overview slides: www.mcc.com/projects/cmi/overview/CMI_Overview/sld001.htm, 1998.
- [Mel98] J. Melton. *Understanding SQL's Stored Procedures*. Morgan Kaufmann Publishers, Inc., 1998.
- [MES95] L. Mummert, R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 143–155, Copper Mountain Resort, Colorado, December 1995.
- [Met96] MetaCase Consulting, Jyväskylä, Finland. *MetaEdit+, Version 2.5 User's Guide*, first edition, November 1996.
- [MGB⁺98] R. McClatchey, J.-M. Le Geoff, N. Baker, W. Harris, and Z. Kovacz. A distributed workflow and product data management application for the construction of large scale scientific apparatus. In Dogac et al. [DÖBS98], pages 18–34.
- [MHL⁺92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), 1992.
- [MHR96] B. Mitschang, T. Härder, and N. Ritter. Design management in CONCORD: Combining transaction management, workflow management and cooperation control. In *Proceedings of RIDE-Interoperability of Non-traditional Database Systems*, pages 160–168, New Orleans, Louisiana, February 1996.
- [Mic98] Microsoft Cooperation. *Deploying Microsoft Exchange Server*. Microsoft Press, 1998.
- [ML92] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 371–380, San Diego, California, June 1992.

- [MN82] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.
- [MO92] L. McGuffin and G. Olson. ShrEdit: A shared electronic workspace. Technical Report 45, University of Michigan, Cognitive Sciences and Machine Intelligence Laboratory, 1992.
- [Moh96] C. Mohan. Concurrency control and recovery methods for B+-tree indexes: ARIES/KVL and ARIES/IM. In Kumar [Kum96], pages 248–306.
- [Moh98] C. Mohan. Recent trends in workflow management products, standards and research. In Dogac et al. [DÖBS98], pages 396–408.
- [Mos82] J. Moss. Nested transactions and reliable distributed computing. In *Proceedings of the IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1982.
- [Mos85] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.
- [Mos87] J. Moss. Nested transactions: An introduction. In B. Bhargava, editor, *Concurrency Control and Reliable Distributed Computing*, pages 395–425. Van Nostrand Reinhold, New York, 1987.
- [MPS91] K. Malcolm, S. Poltrock, and D. Schuler. Industrial strength hypermedia: Requirements for large engineering enterprises. In *Proceedings of the ACM Conference on Hypertext*, pages 13–24, San Antonio, Texas, December 1991.
- [MR93] P. Muth and T. Rakow. VODAK open nested transactions - Visualizing database internals. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 556–557, Washington DC., May 1993. Description of Demonstration.
- [MR97] C. P. Martin and K. Ramamritham. Delegation: Efficiently rewriting history. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 266–275, Birmingham, U.K., April 1997.
- [MRKN92] P. Muth, T. Rakow, W. Klas, and E. J. Neuhold. A transaction model for an open publication environment. In Elmagarmid [Elm92], chapter 6, pages 159–218.
- [MRW⁺93] P. Muth, T. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic concurrency control in object-oriented database systems. In *Proceedings of the 9th IEEE International Conference on Data Engineering (ICDE)*, pages 233–242, Vienna, Austria, April 1993.
- [Mul89] S. Mullender, editor. *Distributed Systems*. ACM Press, 1989.

- [Mut94] P. Muth. *Transaktionsverwaltung in heterogenen und autonomen Datenbanksystemen*. PhD thesis, Darmstadt University of Technology, 1994.
- [Nel81] T. Nelson. *Literary Machines*. Mindful Press, California, 1981.
- [NF97] N. Neves and W. K. Fuchs. Adaptive recovery for mobile environments. *Communications of the ACM*, 40(1):68–74, January 1997.
- [NKCM90] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris. Issues in the design of computer support for co-authoring and commenting. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 183–195, Los Angeles, California, October 1990.
- [NKCM92] C. Neuwirth, D. Kaufer, R. Chandhok, and J. Morris. Flexible diff-ing in a collaborative writing system. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 147–162, Toronto, Ontario, November 1992.
- [NKCM94] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris. Computer support for distributed collaborative writing: Defining parameters of interactions. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 145–152, Chapel Hill, NC, October 1994.
- [NRZ92] M. H. Nodine, S. Ramaswamy, and S. B. Zdonik. A cooperative transaction model for design databases. In Elmagarmid [Elm92], chapter 3, pages 53–85.
- [NSZ91] M. H. Nodine, A. H. Skarra, and S. B. Zdonik. Synchronisation and recovery in cooperative transactions. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 329–341, 1991.
- [NZ90] M. H. Nodine and S. B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, pages 83–94, Brisbane, Australia, August 1990.
- [NZ96] M. H. Nodine and S. B. Zdonik. The impact of transaction management on object-oriented multidatabase views. In Bukhres and Elmagarmid [BE96], chapter 3, pages 57–104.
- [Obe82] R. Obermark. Distributed deadlock detection algorithms. *ACM Transactions on Database Systems*, 7(2):187–210, June 1982.
- [O'N86] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, 1986.
- [ÖV91] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

- [Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.
- [Pap86] C. H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.
- [Pau94] L. C. Paul. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
- [PK92] A. Prakash and M. J. Knister. Undoing actions in collaborative work. In *Proceedings of the Fourth ACM Conference on Computer-Supported Cooperative Work*, pages 273–280, Toronto, Canada, October 1992.
- [PK94] A. Prakash and M. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1(4):295–330, 1994.
- [PKH88] C. Pu, G. E. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Databases (VLDB)*, pages 26–37, Los Angeles, California, USA, August 1988.
- [PS98] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.
- [PTV96] J. Puustjärvi, H. Tirri, and J. Veijalainen. Managing overlapping transactional workflows. In *Proceedings of the 8th Conference on Advanced Information Systems Engineering CAiSE*, pages 24–30, Heraklion, Crete, Greece, May 1996.
- [PTV97] J. Puustjärvi, H. Tirri, and J. Veijalainen. Reusability and modularity in transactional workflows. *Information Systems*, 22(2/3):101–120, April/May 1997.
- [Rak97] T. Rakow. *Semantische Concurrency Control in objektorientierten Datenbanksystemen*. PhD thesis, Darmstadt University of Technology, 1997. Appeared as GMD-Bericht Nr. 284.
- [RB94] M. Rusinkiewicz and M. Bregolin. Specification and execution of transactional workflows in distributed systems. In *Proceedings of the Third Conference on Intelligent Information Systems*. Polish Academy of Sciences, June 1994.
- [RC97] K. Ramamritham and P. K. Chrysanthis. *Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Executive Briefing. IEEE Computer Society Press, 1997.
- [RD98] M. Reichert and P. Dadam. *Adept_{flex}* - Supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
- [Ree83] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

- [RELL90] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin. Extending the transaction model to capture more meaning. *ACM SIGMOD Record*, 19(1), 1990.
- [Reu89] A. Reuter. ConTracts: A means for extending control beyond transaction boundaries. In *Proceedings of the Second International Workshop on High Performance Transaction Systems*, Asilomar, September 1989.
- [RG96] M. Roseman and S. Greenberg. Building real-time groupware with Group-Kit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(1):66–106, 1996.
- [RGN90] T. Rakow, J. Gu, and E.J. Neuhold. Serializability in object-oriented database systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 112–120, Los Angeles, California, 1990.
- [RHD98] M. Reichert, C. Hensinger, and P. Dadam. Supporting adaptive workflows in advanced application environments. In *Proceedings of the EDBT Workshop on Workflow Management Systems*, Valencia, Spain, March 1998.
- [Rit97] N. Ritter. Group-authoring in CONCORD: A DB-based approach. In *Proceedings of the ACM Symposium on Applied Computing*, San Jose, California, February 1997.
- [RKT⁺95a] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a cooperative transaction model: The cooperative activity model. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 194–205, Zurich, Switzerland, September 11–15 1995.
- [RKT⁺95b] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a cooperative transaction model – The cooperative activity model. Technical Report, Arbeitspapiere der GMD 899, Gesellschaft für Mathematik und Datenverarbeitung, March 1995.
- [RM89] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, pages 337–346, Amsterdam, The Netherlands, August 1989.
- [RMH⁺94] N. Ritter, B. Mitschang, T. Härder, M. Gesmann, and H. Schöning. Capturing design dynamics – The CONCORD approach. In *Proceedings of the 10th IEEE International Conference on Data Engineering (ICDE)*, pages 440–451, Houston, Texas, February 1994.
- [RMH96] N. Ritter, B. Mitschang, and T. Härder. Conflict management in CONCORD. In *Proceedings of the Fifth International Conference on Data and Knowledge Systems in Manufacturing and Engineering (DKSME '96)*, pages 81–199, Phoenix, Arizona, October 24–25 1996.

- [RNL95] T. Rakow, E. J. Neuhold, and M. Löhr. Multimedia Database Systems - The Notions and the Issues. In *Tagungsband GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 1–29, Dresden, Germany, March 1995.
- [Roc75] M. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1:364–370, 1975.
- [Rod91] T. Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, December 1991.
- [RS95a] A. Reuter and F. Schwenkreis. ConTracts - A low-level mechanism for building general-purpose workflow management-systems. *IEEE Data Engineering Bulletin*, 18(1):4–10, 1995.
- [RS95b] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In Kim [Kim95], chapter 29, pages 592–620.
- [RSS97] A. Reuter, K. Schneider, and F. Schwenkreis. ConTracts revisited. In Jajodia and Kerschberg [JK97], chapter 5, pages 127–151.
- [Sal96] T. Salzrieder. Design and implemenation of a merger component for cooperative transactions. Diplomarbeit, Fachhochschule Konstanz / GMD-IPSI, Germany, September 1996.
- [SANR92] A. Sheth, M. Ansari, L. Ness, and M. Rusinkiewicz. Using flexible transactions to support multidatabase applications. In *Proceedings of the US West – NSF – DARPA Workshop on Heterogeneous Databases and Semantic Interoperability*, Boulder, Colorado, USA, February 1992.
- [Sar98] M. Sarabadani. Design und Implementierung einer Graphischen Benutzungsoberfläche für ein Kooperatives Mehrbenutzer-System. Diplomarbeit, Darmstadt University of Technology / GMD-IPSI, Germany, Januar 1998.
- [Sat93] M. Satyanarayanan. Mobile computing. *IEEE Computer*, 26(9), September 1993.
- [Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 1–7, Philadelphia, PA, May 1996.
- [SBF⁺87] M. Stefik, D. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems*, 5(2):147–167, April 1987.
- [SCG⁺89] R. Schwanke, E. Cohen, R. Gluecker, W. Hasling, D. Soni, and M. Wagner. Configuration management in the Biin SMS. In *Proceedings of the 11th International Conference on Software Engineering*, pages 383–393, Pittsburgh, PA, May 1989.

- [SCG96] H. Skaf, F. Charoy, and C. Godart. Maintaining consistency of cooperative software development activities. In *Proceedings of the 6th International Workshop on Foundations of Models and Languages for Data and Objects*, pages 103–118, Schloss Dagstuhl, Germany, September 1996.
- [Sch93] F. Schwenkreis. APRICOTS—A Prototype Implementation of a ConTract System—Management of the Control Flow and the Communication System. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, 1993.
- [Sch95] C. Schuckmann. Variable Synchronisationsmechanismen in dem kooperativen Hypermediasystem DOLPHIN. Diplomarbeit, Darmstadt University of Technology / GMD-IPSI, Germany, May 1995.
- [SE98a] D. Spelt and S. Even. An engineering approach to atomic transaction verification: Use of a simple object model to achieve semantics-based reasoning at compile-time. CTIT Technical Report 98-17, University of Twente, The Netherlands, August 1998.
- [SE98b] C. Sun and C. Ellis. Operational transformation in real-time groupware editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 59–69, Seattle, Washington, November 1998.
- [SE99] D. Spelt and S. Even. A theorem prover-based analysis tool for object-oriented databases. In *Proc. of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, Amsterdam, The Netherlands, March 1999.
- [SG88] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.
- [SGHH94] N. Streitz, J. Geißler, J. Haake, and J. Hol. DOLPHIN: Integrated meeting support across liveboards, local and remote desktop environments. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work (CSCW '94)*, pages 345–358, Chapel Hill, N.C., October 1994.
- [SH93] H. Schütt and J. Haake. Server support for cooperative hypermedia systems. In *Proceedings of the Hypermedia Konferenz*, pages 45–56, Zürich, Switzerland, 1993.
- [She97] A. Sheth. From contemporary workflow process automation to adaptive and dynamic work activity coordination and collaboration. In *Proceedings of the DEXA Workshop on Workflow Management in Scientific and Engineering Applications*, Toulouse, France, September 1997.

- [SHH⁺92] N. Streitz, J. Haake, J. Hannemann, A. Lemke, W. Schuler, H. Schütt, and M. Thüring. SEPIA: A cooperative hypermedia authoring environment. In *Proceedings of the Fourth ACM Conference on Hypertext*, pages 11–22, Milano, Italy, November 30 – December 4 1992.
- [SHT89] N. Streitz, J. Hannemann, and M. Thüring. From ideas and arguments to hyperdocuments: Travelling through activity spaces. In *Proceedings of the Second ACM Conference on Hypertext*, pages 343–364, Pittsburgh, PA, November 1989.
- [Sim81] H. A. Simon. *The Sciences of the Artificial*. The MIT Press, Cambridge, Massachusetts, 1981.
- [SK93] P. D. Skopp and G. E. Kaiser. Disconnected operation in a multi-user software development environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, Princeton, NJ, October 1993.
- [SK98] A. Sheth and K. Kochut. Workflow applications to research agenda: Scalable and dynamic work coordination and collaboration systems. In Dogac et al. [DÖBS98], pages 35–60.
- [Ska89] A. H. Skarra. Concurrency control for cooperating transactions in an object-oriented database. *ACM SIGPLAN Notices*, 24(4):145–147, April 1989.
- [Ska91] A. H. Skarra. Localized correctness specifications for cooperating transactions in an object-oriented database. *Office Knowledge Engineering*, 4(1):79–106, February 1991.
- [SKM⁺96a] M. Satyanarayanan, J. Kistler, L. Mummert, M. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile computing environment. In Imieliński and Korth [IK96a], pages 537–570.
- [SKM⁺96b] A. Sheth, K. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shevchenko. Supporting state-wide immunization tracking using multi-paradigm workflow technology. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, pages 263–273, Bombay, India, September 1996.
- [SKSH96] C. Schuckmann, L. Kirchner, J. Schmmer, and J. Haake. Designing object-oriented synchronous groupware with COAST. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 30–38, Boston, Massachusetts, November 1996.
- [SR93] A. Sheth and M. Rusinkiewicz. On transactional workflows. *IEEE Data Engineering Bulletin*, 16(2):37–40, June 1993.
- [SR96] F. Schwenkreis and A. Reuter. Synchronizing long-lived computations. In Kumar [Kum96], pages 336–355.

- [SRK92] A. P. Sheth, M. Rusinkiewicz, and G. Karabatis. Using polytransactions to manage interdependent data. In Elmagarmid [Elm92], chapter 14, pages 556–581.
- [SRN93] A. Storr, U. Rembold, and B. O. Nnaji. *Computer Integrated Manufacturing and Engineering*. Addison-Wesley Publishing Company, 1993.
- [SS84] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.
- [SS90] H. Schütt and N. Streitz. HyperBase: A hypermedia engine based on a relational database management system. In *Proceedings of the First European Conference on Hypertext*, pages 95–108, Versailles, France, November 1990.
- [Str95] N. Streitz. Designing hypermedia: A collaborative activity. *Communications of the ACM*, 38(8):70–71, 1995.
- [STTB99] J. Schümmer, T. Tesch, D. Tietze, and A. Bapat. Introducing groupware in administrative environments – An experience report from the PoliWork project. In *Proceedings of the 8th International Conference on Human-Computer Interaction (HCI)*, Munich, Germany, August 1999. To appear.
- [SWY93] H.-J. Schek, G. Weikum, and H. Ye. Towards a unified theory of concurrency control and recovery. In *Proceedings of the 12th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 300–311, Washington, D.C., May 1993.
- [SYZC97] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. In *Proceedings of the ACM Conference on Supporting Group Work (Group'97)*, pages 425–434, Phoenix, Arizona, November 1997.
- [TBR98] D. Tietze, A. Bapat, and R. Reinema. Document-centric groupware for distributed governmental agencies. In *Proceedings of the 10th Conference on Advanced Information Systems Engineering (CAiSE'98)*, pages 173–190, Pisa, Italy, June 1998.
- [TC94] TRANSCOOP: Transaction management support for cooperative applications – Technical annex part 1: Description of the project, programme of work. TRANSCOOP Consortium, March 1994. Esprit Basic Research Action, Project P8012.
- [TDP⁺94a] D. Terry, A. Demers, K. Peterson, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, Austin, Texas, September 1994.
- [TDP⁺94b] M. Theimer, A. Demers, K. Peterson, M. Spreitzer, D. Terry, and B. Welch. Dealing with tentative data values in disconnected work groups. In *Proceedings*

- of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 192–195, Santa Cruz, California, December 1994.
- [THH91] M. Thüring, J. Haake, and J. Hannemann. What's Eliza doing in the chinese room? Incoherent hyperdocuments and how to avoid them. In *Proceedings of the Third ACM Conference on Hypertext*, pages 161–167, San Antonio, Texas, San Antonio, Texas 1991.
- [THH95] M. Thüring, J. Hannemann, and J. Haake. Hypermedia and cognition: Designing for comprehension. *Communications of the ACM*, 38(8):57–66, 1995.
- [THS90] M. Thüring, J. Hannemann, and N. Streitz. Schreiben als Designproblem: Ein integrativer Ansatz. In *Bericht über den 37. Kongress der Deutschen Gesellschaft für Psychologie*, Göttingen, 1990.
- [Tic85] W. F. Tichy. RCS: A system for version control. *Software Practice and Experience*, 15(7):637–654, 1985.
- [TK96] H. Thimm and W. Klas. Playout management in multimedia database systems. In K. C. Nwozu, P. B. Berra, and B. Thuraisingham, editors, *Design and Implementation of Multimedia Database Management Systems*. Kluwer Academic Publishers, 1996.
- [TTP⁺95] D. Terry, M. Theimer, K. Peterson, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 172–182, Copper Mountain Resort, Colorado, December 1995.
- [TV95] J. Tang and J. Veijalainen. Enforcing inter-task dependencies in transactional workflows. In *Proceedings of the Third International Conference on Cooperative Information Systems*, pages 72–86, Vienna, Austria, May 9–12 1995.
- [TV99] T. Tesch and M. Volz. A lightweight object manager for group-aware applications. GMD Report 47, GMD – Forschungszentrum Informationstechnik GmbH, January 1999.
- [TVF⁺95] T. Tesch, P. Verkoulen, editors, F. Faase, J. Juopperi, A. Lehtola, T. Lillquist, V. Mikonnen, P. Oude Egberink, O. Pihlajamaa, J. Puustjärvi, B. Selders, A. Sladek, J. Veijalainen, J. Wäsch, and A. Wolski. Requirements for the TRANSCOOP transaction model. Report DII.2, Esprit-III LTR Project TRANSCOOP (P8012), January 1995.
- [TVL⁺98] T. Tesch, P. Verkoulen, A. Lehtola, J. Veijalainen, O. Pihlajamaa, and A. Sladek. Application requirements. In de By et al. [dBKV98], chapter 4, pages 59–91.
- [TW95a] T. Tesch and J. Wäsch. Analysis of application scenarios: Cooperative authoring of hyperdocuments. In Tesch and Verkoulen [TVF⁺95], report 3.

- [TW95b] T. Tesch and J. Wäsch. Anforderungen und Modellierung kooperativer Aktivitäten. *Datenbank-Rundbrief, Mitteilungsblatt der GI-Fachgruppe Datenbanken*, Ausgabe 15, May 1995.
- [TW95c] T. Tesch and J. Wäsch. Transaction support for cooperative hypermedia document authoring: A study on requirements. In *Proceedings of 8th ERCIM Database Research Group Workshop on Database Issues and Infrastructure in Cooperative Information Systems*, pages 31–42, Trondheim, Norway, August 23–25 1995.
- [TW97] T. Tesch and J. Wäsch. Global Nested Transaction Management for ODMG-Compliant Multi-Database Systems. In *Proceedings of the Sixth International Conference on Information and Knowledge Management (CIKM)*, pages 67–74, Las Vegas, Nevada, November 10–14 1997.
- [US92] R. Unland and G. Schlageter. A transaction manager development facility for non standard database systems. In Elmagarmid [Elm92], chapter 11, pages 399–466.
- [VdBA98] J. Veijalainen, R. de By, and K. Aberer. The TRANSCOOP paradigm. In de By et al. [dBKV98], chapter 2, pages 11–25.
- [VEH89] J. Veijalainen, F. Eliassen, and B. Holtkamp. Concurrency control and object-oriented databases. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 16, pages 395–421. ACM Press, 1989.
- [VEH92] J. Veijalainen, F. Eliassen, and B. Holtkamp. The S-transaction model. In Elmagarmid [Elm92], chapter 12, pages 467–513.
- [Ver78] J. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–196, 1978.
- [vEVD89] P. H. J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. Elsevier Science Publishers B.V., 1989.
- [VFSE95] P. Verkoulen, F. Faase, A. Selders, and P. Oude Egberink. Requirements for an advanced database transaction model to support design for manufacturing. In *Proceedings of the Flexible Automation and Intelligent Manufacturing Conference*, pages 102–113, Stuttgart, Germany, June 1995.
- [VHBS98] R. Vingralek, H. Hasse, Y. Breitbart, and H.-J. Schek. Unifying concurrency control and recovery of transactions with semantically rich operations. *Journal of Theoretical Computer Science*, 190(2):363–396, 1998.
- [vKSA⁺95] M. van Keulen, J. Skowronek, P. Apers, H. Balsters, H. Blanken, R. de By, and J. Flokstra. A framework for representation, validation and implementation of database application semantics. In *Proceedings of the Sixth IFIP Conference on Database Semantics (DS-6)*, Atlanta, Georgia, May 30 – June 2 1995.

- [VPL95] J. Veijalainen, O. Pihlajamaa, and A. Lehtola. Research issues in workflow systems. In *Proc. of 8th ERCIM Database Research Group Workshop on Database Issues and Infrastructure in Cooperative Information Systems*, pages 43–61, Trondheim, Norway, August 1995.
- [VRS86] S. Vinter, K. Ramamritham, and D. Stemple. Recoverable actions in Gutenberg. In *Proceedings of the Sixth IEEE International Conference on Distributed Computing Systems*, pages 242–249, Washington, D.C., May 1986.
- [VTF⁺95] P. Verkoulen, T. Tesch, editors, F. Faase, J. Juopperi, A. Lehtola, T. Lillquist, V. Mikonnen, P. Oude Egberink, O. Pihlajamaa, J. Puustjärvi, B. Selders, A. Sladek, J. Veijalainen, J. Wäsch, and A. Wolski. Requirements for the TRANSCOOP specification language. Report DII.1, Esprit-III LTR Project TRANSCOOP (P8012), January 1995. Also appeared as Memoranda Informatica 95-16, University of Twente, The Netherlands, February 1995.
- [VWP⁺98] J. Veijalainen, J. Wäsch, J. Puustjärvi, H. Tirri, and O. Pihlajamaa. Transaction models in cooperative work – An overview. In de By et al. [dBKV98], chapter 3, pages 27–58.
- [WA95] J. Wäsch and K. Aberer. Flexible design and efficient implementation of a hypermedia document database system by tailoring semantic relationships. In *Proceedings of the Sixth IFIP Conference on Database Semantics (DS-6)*, Atlanta, Georgia, May 30 – June 2 1995.
- [WA97] J. Wäsch and K. Aberer. Flexible design and efficient implementation of a hypermedia document database system by tailoring semantic relationships. In R. Meersman and L. Mark, editors, *Database Applications Semantics*, pages 367–387. Chapman & Hall, 1997. (Revised version of [WA95]).
- [WAN98] J. Wäsch, K. Aberer, and E. J. Neuhold. Transactional support for cooperative applications. In Dogac et al. [DÖBS98], pages 304–338.
- [Wäs96] J. Wäsch. History merging as a mechanism for information exchange in cooperative and mobile environments. *Datenbank-Rundbrief, Mitteilungsblatt der GI-Fachgruppe Datenbanken*, Ausgabe 17, May 1996.
- [Wäs97] J. Wäsch. Comparison of the extensibility infrastructure of commercial object-relational DBMS. Report prepared for Software AG, Darmstadt, Germany, December 1997.
- [WBHN87] J. Walpole, G. Blair, D. Hutchison, and J. Nicol. Transaction management for distributed programming environments. *Software Engineering Journal*, 2(5):169–177, 1987.
- [WBMN88] J. Walpole, G. Blair, J. Malik, and J. Nicol. Maintaining consistency in distributed software engineering environments. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 418–425, June 1988.

- [WC95] G.D. Walborn and P.K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14th Symposium on Reliable Distributed Computing*, Bad Neuenahr, Germany, September 1995.
- [WC97] G. Walborn and P. Chrysanthis. PRO-MOTION: Support for mobile database access. *Personal Technologies Journal*, 1(3), September 1997.
- [WDSS93] G. Weikum, A. Deacon, W. Schaad, and H. Schek. Open nested transaction in federated database systems. *IEEE Data Engineering Bulletin*, 16(2):4–7, June 1993.
- [WEF⁺96] J. Wäsch, editor, S. Even, F. Faase, H. Kaijanranta, J. Klingemann, A. Lehtola, O. Pihlajamaa, and T. Tesch. Design of the TRANSCOOP demonstrator system. Report DVII.1, Esprit-III LTR Project TRANSCOOP (P8012), October 1996.
- [Wei86] G. Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 31–42, Cambridge, Massachusetts, March 1986.
- [Wei88a] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.
- [Wei88b] G. Weikum. *Transaktionen in Datenbank-Systemen*. Addison-Wesley Publishing Company, 1988.
- [Wei89a] W. E. Weihl. The impact of recovery on concurrency control. In *Proceedings of the 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 259–269, 1989.
- [Wei89b] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, 1989.
- [Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, 1991.
- [Wei93] W. E. Weihl. The impact of recovery on concurrency control. *Journal of Computer and System Sciences*, 47(1):157–184, 1993.
- [Wes98] M. Weske. Flexible modeling and executions of workflow activities. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, 1998.
- [WfM96a] WfMC. Workflow reference model. Technical Report WFMC-TC00-1003, Workflow Management Coalition, November 1996. Issue 1.1.
- [WfM96b] WfMC. Workflow standard - Terminology & glossary. Technical Report WFMC-TC-1011, Workflow Management Coalition, June 1996. Version 2.0.

- [WfM98] WfMC. Workflow and Internet - Catalyst for radical change. White paper, Workflow Management Coalition, June 1998.
- [WH93] G. Weikum and C. Hasse. Multilevel transaction management for complex objects: Implementation, performance, parallelism. *VLDB Journal*, 2(4):407–453, October 1993.
- [WHBM90] G. Weikum, C. Hasse, P. Brössler, and P. Muth. Multi-level recovery. In *Proceedings of the 9th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1990.
- [Wii91] U. Wiil. Using events as support for data sharing in collaborative work. In *Proceedings of the International Workshop on CSCW*, pages 162–177, Berlin, Germany, April 1991.
- [Wii92] U. Wiil. Issues in the design of EHTS: A multiuser hypertext system for collaboration. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 629–639, Kauai, Hawaii, January 1992.
- [Wii93] U. Wiil. Experiences with HyperBase: A hypertext database supporting collaborative work. *ACM SIGMOD Record*, 22(4):19–25, 1993.
- [WK96] J. Wäsch and W. Klas. History merging as a mechanism for concurrency control in cooperative environments. In *Proceedings of RIDE—Interoperability of Non-traditional Database Systems*, pages 76–85, New Orleans, Louisiana, February 26–27 1996.
- [WL85] W. E. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, March 1985.
- [WL92] U. Wiil and J. Leggett. Hyperform: Using extensibility to develop, dynamic, open and distributed hypertext systems. In *Proceedings of the ACM Conference on Hypertext*, pages 251–261, Milano, Italy, December 1992.
- [WL93] U. Wiil and J. Leggett. Concurrency control in collaborative hypertext systems. In *Proceedings of the Fifth ACM Conference on Hypertext*, pages 14–18, Seattle, Washington, November 14–18 1993.
- [WO90] U. Wiil and K. Oesterby. Experiences with HyperBase – A multi-user back-end for hypertext with emphasis on collaboration support. Technical Report R 90-38, Aalborg University, Denmark, October 1990.
- [WR90] H. Wächter and A. Reuter. Grundkonzepte und Realisierungsstrategien des ConTract-Modells. *Informatik Forschung und Entwicklung*, 5:202–212, 1990.
- [WR92] H. Wächter and A. Reuter. The ConTract model. In Elmagarmid [Elm92], chapter 7, pages 219–264.

- [WS92] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In Elmagarmid [Elm92], chapter 13, pages 515–554.
- [WS97] D. Worah and A. Sheth. Transactions in transactional workflows. In Jajodia and Kerschberg [JK97], chapter 1, pages 3–33.
- [YAM89] E. Yoderer, R. Aksyn, and D. McCracken. Collaboration in KMS, a shared hypermedia system. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 465–470, Austin, Texas, Mai 1989.
- [YHMD88] N. Yankelovich, B. Haan, N. Meyrowitz, and S. Drucker. Intermedia: The concept and the construction of seamless information environments. *IEEE Computer*, 21(1):36–51, January 1988.

Appendix A: List of Figures

2.1	Spectrum of Investigated Classes of Cooperative Applications.	20
2.2	Example of Open-Nested Transactions.	28
2.3	Example of Splitting and Joining Transactions.	30
2.4	Example of Check-Out/Check-In.	32
3.1	Workspaces and Exchange Facilities in the COACT Model.	43
3.2	Backward Commutativity.	63
3.3	Forward Commutativity.	73
3.4	Relationships among the Different Commutativity Relations.	83
3.5	The Graphical User Interface of the DBMS-based SEPIA System.	87
3.6	Sample SEPIA Cooperative Activity.	95
3.7	Computation of Consistent Units of Work with Backward Commutativity. . . .	97
3.8	Computation of Minimally Closed Subhistory with Backward Commutativity. .	99
3.9	Independent Closed Subhistories.	99
3.10	Example for Merged Histories.	103
3.11	Incremental Merge with $bc \neq fc$	104
3.12	Anomaly resulting from $bc \neq fc$	105
3.13	Example of Correctly Merged Histories.	108
3.14	Conservative Compensation.	113
3.15	Selective Compensation.	116
3.16	Situation before Delegation from Workspace A to Workspace B.	122
3.17	Situation after Delegation from Workspace A to Workspace B.	123
3.18	Situation before Import from Workspace B into Workspace A.	124
3.19	Situation after Import from Workspace B into Workspace A.	125
3.20	Situation before/after Save from Workspace A to the Common Workspace. . . .	126
3.21	Situation after Import from the Common Workspace into Workspace B.	127
3.22	Sub-Partitioning of Exchange Partitions.	130
3.23	Graphical Representation of the Sample CoCoA Scenario WriteProposal. . . .	139
3.24	Sketch of a Sample CoCoA Specification for Cooperative Authoring.	140
3.25	Relationship of the COACT Model to Selected Advanced Transaction Models. .	146
4.1	General Architecture of Mobile Computing Systems.	150

4.2	Different Kinds of Connectivity in Mobile Environments.	152
4.3	Extended COACT Architecture and Information Exchange Options.	154
4.4	Modes of Operation in the Mobile COACT Architecture.	158
5.1	Architecture of the TRANSCOOP Demonstrator System.	166
5.2	Simplified Architecture of the VODAK Object-Oriented DBMS.	171
5.3	Client-Server Architecture of the VODAK-Based SEPIA Implementation. . . .	173
5.4	Sample Interface of the Metaclass HME_Node_MC.	174
5.5	Sample Application Classes of the VML Database Schema for SEPIA.	175
5.6	The Graphical User Interface of the SEPIA Frontend.	176
5.7	Sample Metaclass Extended with VML Methods for Compensation.	177
5.8	The Runtime Architecture of the TRANSCOOP Demonstrator System.	178
5.9	Interaction of the Different Components in the Runtime System.	182
5.10	Pull-Down Menus for Cooperation Functionality.	184
5.11	Selecting Operations for Import.	185
5.12	Analyzing Merge Conflicts.	186
5.13	Undoing Operations.	187

Appendix B: List of Tables

2.1	Comparison of Advanced Transaction Models.	37
3.1	Backward Commutativity Relation for ACCOUNTS.	64
3.2	Backward Commutativity Relation for ACCOUNTS-2.	66
3.3	Right Backward Commutativity Relation for ACCOUNTS-2.	70
3.4	Forward Commutativity Relation for ACCOUNTS-2.	74
3.5	Left Forward Commutativity Relation for ACCOUNTS-2.	78
3.6	Invocation-based Commutativity Relation for ACCOUNTS-2.	82
3.7	Backward Commutativity Relation for the SEPIA Cooperative Activity Type. .	89
3.8	Forward Commutativity Relation for the SEPIA Cooperative Activity Type. . .	90
4.1	Characteristics of Mobile Environments.	151
4.2	Serving Cache Misses and Update Propagation in Different Operation Modes. .	159

Appendix C: List of Examples

Example 1: Software development example	21
Example 2: Relaxed atomicity (requirement (A))	22
Example 3: Support for retraction of decisions (requirement (B))	23
Example 4: Support for consistent exchange of work (requirement (C))	23
Example 5: Support for private and shared data (requirement (D))	24
Example 6: Coordination of individual and joint work (requirement (E))	24
Example 7: Semantics-based concurrency control	26
Example 8: Open-nested transactions	28
Example 9: Split/join transactions	29
Example 10: Check-out/check-in	31
Example 11: Bank accounts	48
Example 12: Operation types for ACCOUNTS	50
Example 13: Operation invocations	52
Example 14: Operations	53
Example 15: Legal and illegal operations	54
Example 16: Histories, subhistories, prefixes, and continuation	56
Example 17: Legal and illegal histories	56
Example 18: Equivalence of states and histories	58
Example 19: Backward commutativity relation for ACCOUNTS database	63
Example 20: Backward commutativity relation for ACCOUNTS-2 database	64
Example 21: Asymmetry of right backward commutativity	68
Example 22: Right backward commutativity relation for ACCOUNTS-2 database	69
Example 23: “False” dependencies of the <i>rbc</i> relation	69
Example 24: Forward commutativity relation for ACCOUNTS-2	74
Example 25: Incomparability of forward and backward commutativity	75
Example 26: Left forward commutativity	77
Example 27: Left forward commutativity relation for ACCOUNTS-2 database	77
Example 28: Invocation-based commutativity relation for ACCOUNTS-2 database	81
Example 29: Compensation function and compensation operation	85
Example 30: Sample invocable operation types for SEPIA	87
Example 31: Compensation function for SEPIA cooperative activity type	88
Example 32: Merging relations for SEPIA cooperative activity type	89

Example 33: SEPIA cooperative authoring activity	94
Example 34: Closed subhistory	97
Example 35: Minimally closed and independent subhistories	99
Example 36: Merged histories	101
Example 37: Incremental merge with $bc \neq fc$	104
Example 38: Merge anomaly resulting from $bc \neq fc$	104
Example 39: Correctly merged history	107
Example 40: Non-serializable correctly merged history	110
Example 41: Compensation operation	111
Example 42: Conservative compensation approach	112
Example 43: Directly compensatable operation	114
Example 44: Selective compensation approach	116
Example 45: Delegation from workspace A to workspace B	122
Example 46: Import from workspace B into workspace A	123
Example 47: Non-serializable correctly merged histories	125
Example 48: Saving to and importing from the common workspace	126
Example 49: Sub-partitioning	129
Example 50: Transparent operations in selective compensation	132
Example 51: Masked operations	133
Example 52: Synchronization points	134

Appendix D: List of Definitions

Definition 1: Data objects	47
Definition 2: Identity of data objects	47
Definition 3: Workspace	47
Definition 4: States	48
Definition 5: Equality of states	48
Definition 6: Operation types	49
Definition 7: Operation invocation	52
Definition 8: Operation	52
Definition 9: Instances of operation invocations	53
Definition 10: Identity of operations	53
Definition 11: Generation and re-execution of operations	53
Definition 12: Legal operation	54
Definition 13: History	55
Definition 14: Equality of histories	55
Definition 15: Subhistory and restriction of a history	55
Definition 16: Prefix, continuation, and reordering of a history	55
Definition 17: Legal history	56
Definition 18: Consistent state	57
Definition 19: Equivalence of states	57
Definition 20: Equivalence of histories	58
Definition 21: Backward commutativity relation	62
Definition 22: Right backward commutativity relation	69
Definition 23: View of an operation	70
Definition 24: Forward commutativity relation	73
Definition 25: Left forward commutativity	77
Definition 26: Model-independent commutativity relation	79
Definition 27: Invocation-based commutativity relation	80
Definition 28: Cooperative activity type	83
Definition 29: Workspaces and workspace histories	90
Definition 30: Cooperative activity	91
Definition 31: Primitives for the management of cooperative activities	92
Definition 32: Exchange meta-operations	93

Definition 33: Closed subhistory	96
Definition 34: Minimally closed subhistory	98
Definition 35: Independent subhistories	99
Definition 36: Mergeable histories	101
Definition 37: Merged history	101
Tentative Definition: Correctly merged history	103
Definition 38: Correctly merged history	106
Definition 39: Compensation operation	111
Definition 40: Compensation function	112
Definition 41: Directly compensatable operation	113
Definition 42: Unit of compensation	114
Definition 43: Minimal unit of compensation	115
Definition 44: Compensation sequence	115
Definition 45: Transparent and read-only operations	130
Definition 46: Noop operation	131
Definition 47: Generalized correctly merged history	135
Definition 48: Recoverability relation	137
Definition 49: Ignores relation	137

Appendix E: List of Lemmas

Lemma 1: Legal prefixes and illegal continuations of histories	57
Lemma 2: Relationship between equality and equivalence of states	58
Lemma 3: Equivalence of states and equivalence of histories are equivalence relations	58
Lemma 4: Symmetry of backward commutativity	63
Lemma 5: Relationship between the <i>bc</i> and <i>rbc</i> relations	69
Lemma 6: Views of operations and <i>rbc</i> relation	71
Lemma 7: Symmetry of forward commutativity	73
Lemma 8: Incomparability of forward and backward commutativity	74
Lemma 9: Relationship between the <i>lfc</i> and <i>fc</i> relations	77
Lemma 10: Symmetry of model-independent commutativity	80
Lemma 11: Symmetry of invocation-based commutativity	81
Lemma 12: Relationship among the <i>ic</i> and <i>c</i> relations	81
Lemma 13: Uniqueness of minimally closed subhistories	98
Lemma 14: Two disjoint, closed subhistories are independent	100
Lemma 15: Neglection of backward conflicts among operations in deferred-update relationship	108
Lemma 16: Conservative compensation	112
Lemma 17: Compensation for directly compensatable operations	113
Lemma 18: Uniqueness of minimal unit of compensation	115
Lemma 19: Read-only operations are transparent operations	131
Lemma 20: Forward commutativity of transparent operations	131
Lemma 21: Left forward commutativity of transparent operations	131
Lemma 22: <i>Noop</i> is compensation operation for transparent operations	131
Lemma 23: Transparent operations need not be re-executed	132

Appendix F: List of Propositions

Proposition 1: Replacement of legal prefixes in legal histories	59
Proposition 2: Reordering of backward commuting operations in legal histories . . .	65
Proposition 3: Serializability for update-in-place models	67
Proposition 4: Reordering of right backward commuting operations in legal histories	71
Proposition 5: Merging disjoint legal histories with forward commutativity	75
Proposition 6: Merging disjoint legal histories with left forward commutativity	78
Proposition 7: Relationships among different commutativity relations	82

Appendix G: List of Theorems

Theorem 1: Closed subhistories of legal histories are legal	96
Theorem 2: Legality and equivalence of correctly merged histories	109
Theorem 3: Legality and equivalence of histories after selective compensation	115
Theorem 4: The basic merge algorithm computes correctly merged histories	120
Theorem 5: Legality and equivalence of generalized correctly merged histories	135

