Semantics-Based Transaction Management for Cooperative Applications

Justus Klingemann, Thomas Tesch, and Jürgen Wäsch

Integrated Publication and Information Systems Institute (GMD-IPSI)
GMD – German National Research Center for Information Technology
Dolivostr. 15, D-64293 Darmstadt, Germany
{klingem,tesch,waesch}@darmstadt.gmd.de
http://www.darmstadt.gmd.de/~{klingem,tesch,waesch}

Proceedings of the International Workshop on Advanced Transaction Models and Architectures (ATMA), Goa, India, August 31 - September 2, 1996, pp. 234-252.

Semantics-Based Transaction Management for Cooperative Applications*

J. Klingemann T. Tesch J. Wäsch
GMD-IPSI Integrated Publication and Information Systems Institute
Dolivostraße 15, D-64293 Darmstadt, Germany
{klingem,tesch,waesch}@darmstadt.gmd.de

Abstract

Cooperative applications require advanced transactional services that make the sharing and exchange of information as natural as possible while preserving a notion of consistency. Conventional transaction models that were based on the ACID properties do not meet the typical requirements of cooperative applications.

The COACT model [RKT+95, WK96] is designed for supporting cooperative work in interactive multi-user environments. In this paper, we utilize semantics-based transaction management concepts to develop a formal execution model for COACT. We present a correctness criterion for the exchange of information between users and discuss our cooperation mechanism in the context of different semantic relations.

1 Introduction

Modern application areas of the transaction concept, such as CAD, CASE, or groupware applications like cooperative authoring impose new requirements on transaction models [Kai95, Kor95]. In all of these non-standard database applications there is a need to support the collaboration of human actors engaged in common tasks and working towards a common goal. A transaction model supporting cooperative applications should make the sharing and exchange of information among co-workers as natural as possible while still preserving a notion of consistency.

In the TRANSCOOP project we investigated several cooperative application domains, namely, cooperative authoring [TW95], design for manufacturing [VFSE95] and workflow applications [JLP+95] to derive requirements for a cooperative transaction model [TV95]. In general, transactions in such environments need to support the *interactive* processing of activities of *long, uncertain duration*, in which competition for resources is replaced by the *need to cooperate*.

The emphasis, therefore, is not on preventing access to resources (like in a serial world), but rather on the interoperation, i.e., the *semantically correct exchange of information*, among concurrent activities of cooperating (possibly geographically distributed or mobile) users. In such environments, failure atomicity may be too strict, and isolation among concurrent users

*This work was done in the ESPRIT project TRANSCOOP (EP8012) which is funded by the Commission of the European Communities. The partners in the TRANSCOOP project are GMD (Germany), University of Twente (The Netherlands), and VTT (Finland).

contradicts the need of cooperation. Hence, we need to replace these criteria by new ones which are more suitable for cooperative applications.

Most approaches found in the CSCW area [EGR91] synchronize cooperative access to shared data in a more or less ad-hoc manner using user-controlled locking, different lock modes, and notifications [WL93, TW95]. Like CSCW systems, most of the approaches in the CAD domain are based on an intuitive model of interaction and do not explicitly address cooperation among co-workers [Kai95].

Semantics-based transaction processing techniques [SS84, Wei88, FÖ89, WS92, MRW+93, WC95] seem to be a promising approach to overcome concurrency limitations of traditional transaction models. These techniques exploit either the semantics of methods defined on objects (data-oriented approach), or the semantics of transactions (transaction oriented approach)[SZ89] to enhance concurrency in object-oriented database management systems (OODBMS).

In the COACT model [RKT+95, WK96], we have proposed to utilize semantics-based transaction management concepts to relax the isolation property of traditional transaction models. COACT provides a flexible framework to support cooperation in interactive, multi-user environments.

In this paper we present a formal treatment of COACT's semantics-based transaction management concepts. We use both *backward* and *forward commutativity* [Wei88, LMWF94] relations to develop a correctness criterion for the exchange of information among co-workers. Moreover, we discuss the usability of semantic relations that go beyond commutativity [Her90, BR92] in the COACT model and examine their impact on the cooperation mechanism.

The rest of the paper is structured as follows. In section 2, we introduce the logical system architecture of COACT. In the subsequent section we describe the constituent components of a specification of a cooperative activity. In section 4, we present the execution formalism of the COACT model. First, we describe how we model the execution of a single user's work. Then, we focus on enabling cooperation in COACT by means of merging histories and present a correctness criterion. Afterwards, we discuss the applicability of the relations *ignores* and *recoverability*. We conclude by presenting related work in the area of cooperative transaction management and work to be done in the future.

2 Overview of the COACT Model

In general, cooperative work is characterized by alternating periods of individual and joint work [TV95, TW95, Kai95]. During individual work periods, users try out alternative problem solutions while co-workers may work simultaneously on the same subject. Access to and use of shared data 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. Moreover, dynamically formed subgroups among co-workers should be possible.

Therefore, we assign in COACT a *private workspace* to every user who takes part in a collaborative effort (*cooperative activity*). By default, the private workspaces of the co-workers are isolated from each other. Additionally, there exists a *common workspace* per cooperative activity. This workspace is isolated from the private workspaces and is not assigned to a single actor in a cooperative activity. The common workspace contains the data items available when a cooperative activity is *started* and the results of the cooperative activity when *committed*. Figure 1 gives

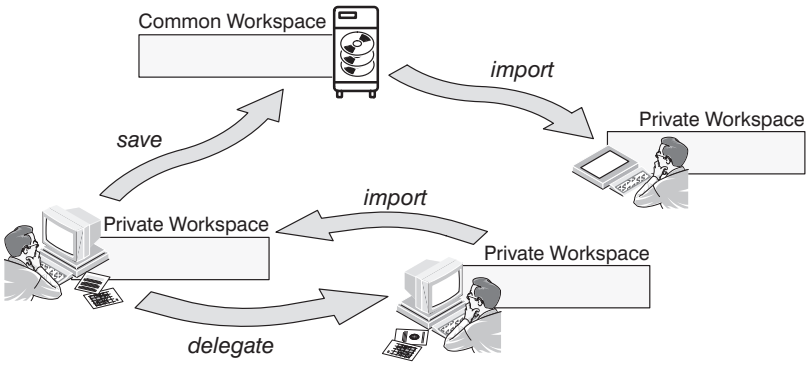

Figure 1: Workspaces and exchange facilities in COACT

an overview of the constituents of the COACT model.

To achieve isolation of workspaces, we (conceptually) copy the data items initially contained in the common workspace to all private workspaces. From these copies the actors can create throughout the working process their private versions of data items that can be manipulated independently. Hence, modifications to data items done by different co-workers do not interfere. For each workspace, we keep a log of the modifications in a *workspace history*.

To model the correct interaction, i.e., cooperation, of users involved in a cooperative activity, we introduce the concept of *cooperative activity types* (*CAT*). Each cooperative activity is specified by a single *CAT*. First, a *CAT* describes a set of constituent activity types that can be invoked by a user in his private workspace. We denote the execution of one of these constituent activities by the term *activity instance*. Second, a *CAT* describes a set of merging rules that exploit the semantics of activity instances to guide the process of 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 operations*. The exchange operations are generic meta operations of the COACT model (like starting, aborting, or committing of a cooperative activity) and are all based on the paradigm of merging workspace histories. The COACT model provides two different options for exchanging information:

1. Co-workers can directly exchange activity instances between their private workspaces by means of *import* and *delegate* operations. The import operation is used by a co-worker to incorporate activity instances executed in the scope of another workspace into his own workspace. The importing user is responsible for resolving conflicts that may occur during the merge. The delegate operation is used to pass on a set of activity instances to a co-worker who is then responsible for merging them into his own workspace.

2. Co-workers can exchange activity instances through the common workspace by means of *save* and *import* operations. An actor can use the save operation to incorporate activity instances of his workspace into the common workspace, thus, making (parts of) his results public to all co-workers. The user who invokes the save is responsible for the resolution of conflicts. Other co-workers can retrieve this information using the import operation described above.

The merge mechanism allows only to exchange consistent parts of workspaces. We identify such consistent units of work by examining the backward commutativity relation [Wei88, LMWF94] between activity instances contained in a workspace history. The incorporation of activity instances is then realized by the re-execution of the activity instances in the respective destination workspace. In this way, the effects of these activity instances are reflected in the private versions of the data items in the destination workspace. The semantic correctness of the exchange of activity instances is guaranteed by ensuring that the re-execution of an activity instance has an equivalent "view" on the history in the destination workspace as in the source workspace. Hence, the behavior of activity instances in terms of output parameter values is indistinguishable from the initial execution. We use the forward commutativity relation [Wei88, LMWF94] to check this. If the merge process cannot be performed without violating the semantical correctness, the merger offers different consistent sets of activity instances. The controlling user then selects one of the offered solutions. To facilitate the merge process for the user, the selection task can alternatively be performed within a software module without requiring user interactions. In this case, certain merge policies can be specified providing different conflict resolution strategies.

If an activity instance has been successfully incorporated into another workspace, it is conceptually the same activity instance which is present in *more than one workspace*. The presence of identical activity instances in several workspaces enables us to establish a close cooperation between co-workers. The degree of cooperation is scalable depending on the exchange frequency.

Those parts of a cooperative activity that are reflected in the common workspace after its completion (commit) are considered as its final result. 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.

3 Specification of cooperative activities

In this section we describe what are the parts of a cooperative activity type (*CAT*) and how they are specified. We model a *CAT* as a tuple (sig, T, M) where

- sig is the signature describing the interface,
- T is a set of activity types,
- and M is a set of merging rules.

In the remainder of this section, the specification of activity types and merging rules are described in detail.

3.1 Activity types

The set T defines a finite set of activity types. The actors of the cooperative activity apply instances of the types in T to their workspaces. Each activity type $t \in T$ either refers to the signature of a CAT or an elementary activity type. Elementary activity instances appear as atomic state changes on the respective workspace and are executed, in contrast to cooperative activities, by a single actor. With Σ we denote the (not necessarily finite) set of all possible instances of the activity types in T.

To enable the exchange of activity instances we require that the specified activity types are deterministic functions and guarantee termination. We further require that for all activity instances a compensating activity instance is specified. The compensating activity semantically undoes effects of the original activity rather than physically restoring the prior workspace state [KLS90, NZ96]. It is the scenario designer's responsibility to specify the compensating activity. Formally, compensation is specified by a function $comp : \Sigma \rightarrow \Sigma$ that assigns to each activity instance the compensating activity instance. Compensation is further described in section 4.6.

To let users act autonomously within their workspaces, we compensate activities only within a single workspace, i.e., if an activity is compensated in one workspace it may still be present in other workspaces.

3.2 Merging rules

The merging rules are needed to allow for a semantically correct exchange of activity instances among workspaces. The merging rules consist of the backward commutativity relation (bc) and the forward commutativity relation (*fc*) [Wei88, LMWF94] defined over activity instances.

The specification of the relations bc and *fc* provides for each pair of activity types a predicate involving the parameters of their signatures. By applying the specified predicates to the respective activity instances, the membership in the bc or *fc* relation, respectively, can be easily checked at runtime. This specification concept corresponds to commutativity tables as introduced in [BBG$^+$83, Kor83, SS84]. Note, the evaluation of the predicate depends only on those parts of the workspace state reflected in the parameters.

The semantics of both relations is defined in sections 4.5 and 4.7. We will discuss the usage of more sophisticated relations and their impacts on the CoAct model in section 4.9.

4 Formal Execution Model

In the following, we define the basic building blocks of our model that allow us to derive our notion of correctness in the remainder of this section.

4.1 States

Let E be the set of all data items that can occur in a workspace. Let $dom(e)$ denote for each data item $e \in E$ its domain. Then, a state S is a mapping that assigns to each $e \in E$ a corresponding value from $dom(e) \cup \{\perp\}$. If S denotes the state of a workspace ws, we define $e \in ws :\Leftrightarrow S(e) \neq \perp$. With $S_\perp$ we denote the empty state, i.e., $\forall e \in E : S(e) = \perp$.

4.2 Activity instances

We model an activity instance $act \in \Sigma$ as a tuple (id, t, I, O) consisting of

1. a unique identifier id,
2. the corresponding activity type t,
3. a set of input parameters I,

4. a set of output parameters O.

To refer to the different elements of an activity instance, we write $act.component$, e.g., $act.t$ to refer to the corresponding activity type.

The unique identifier id is needed to identify *identical activity instances* that occur in more than one workspace. This is not possible via parameters only. The scope of uniqueness is the corresponding cooperative activity, i.e., if an activity instance is exchanged among workspaces its unique identifier remains untouched such that it denotes the initial (first) execution although it is re-executed in the destination workspace.

The parameters can either contain data items from the workspace state or values provided by the invoking user. Note, an activity instance is allowed to access additional data items not reflected in its signature.

Output parameters are enclosed in our concept of activity instances, too. Hence, an activity instance models not only an invocation but also its corresponding response event [Wei88]. Therefore, we first have to define whether invocation and response match for a certain state. This is captured by our notion of legal activity instances.

Definition 1 (legal activity instance) An activity instance $act = (id, t, I, O)$ is *legal* in a state S iff $act.t$ applied to state S with input parameters $act.I$ yields output parameters $act.O$.

4.3 Histories

A history $h = (ACT, \ll_{ACT})$ is a finite set $ACT \subseteq \Sigma$ of activity instances together with a total order $\ll_{ACT}$ over ACT reflecting the execution order. For simplicity, we denote a history $h = (ACT, \ll_{ACT})$ as $[a_1, \ldots, a_n]$ iff $ACT = \{a_1, \ldots, a_n\} \wedge (a_i, a_j) \in \ll_{ACT} \Leftrightarrow i < j$. Such sequences can be concatenated, i.e., $[a_1, \ldots, a_i, a_j, \ldots, a_n] = [a_1, \ldots a_i] \bullet [a_j, \ldots, a_n]$. Next, we extend the notion of legal activity instances to legal histories by applying the above definition to finite execution sequences.

Definition 2 (legal history) A history $[a_1, \ldots, a_n]$ is legal for a state S iff there is a sequence of states $S_0, \ldots, S_{n\perp 1}$ with $S = S_0$, a_i legal in $S_{i\perp 1}$, and S_i results from the execution of a_i in state $S_{i\perp 1}$ for $i = 1, \ldots, n$. We say a history is legal if it is legal for state $S_\perp$.

With $S_n = [a_1, \ldots, a_n](S_0)$ we refer to the state S_n resulting from the execution of the sequence $[a_1, \ldots, a_n]$ in state S_0. The notion of legal histories allows us to define the equivalence of states. Two states are equivalent if they are not distinguishable for all subsequent activity instances.

Definition 3 (equivalent state) Two states S and S' are *equivalent* ($S \equiv S'$) iff

$$\forall \text{ histories } h : h(S) \text{ legal } \Leftrightarrow h(S') \text{ legal}$$

With $h_1 \equiv h_2$ we refer to $h_1(S_\perp) \equiv h_2(S_\perp)$.

Definition 4 (subhistory) A history $h' = (ACT', \ll_{ACT'})$ is a *subhistory* of $h = (ACT, \ll_{ACT})$ iff

$$ACT' \subseteq ACT \wedge$$
$$\forall a', b' \in ACT' : (a', b') \in \ll_{ACT} \Leftrightarrow (a', b') \in \ll_{ACT'}$$

4.4 Workspace histories

A cooperative activity CA is a set of histories representing the workspaces with an identical initial workspace state S_{init}. Therefore, we write $CA = (S_{init}, \{WS_0, WS_1, \ldots, WS_n\})$. The work of a single user within the scope of a cooperative activity is modeled as a *workspace history*. The workspace WS_0 denotes the common workspace of the cooperative activity. In practise, the number of workspaces can change dynamically as participants may join or leave the cooperative activity.

Conceptually, the state S_{init} is *copied* to each workspace to achieve isolation of executions in different workspaces, e.g., by a dynamic replication scheme. Hence, activities can be executed independently in different workspaces. To ensure that subsequent definitions are independent of S_{init}, we require $\exists$ history $h : h(S_\perp) = S_{init}$.

4.5 Dependencies within a single history

So far we have explained how actors work individually in their private workspaces. Now, we will describe how activity instances can be exchanged among workspaces. First of all, we introduce the central concept of *consistent units of work*.

An activity instance might depend on previously executed activity instances, i.e., its behavior is influenced by these activity instances. Hence, we may not be able to exchange a single activity instance but have to take care that all activity instances it depends on are exchanged, too. By identifying all relevant predecessors of a given activity instance, we are able to identify consistent units of work that can be subject of an information exchange.

Since we are interested here in determining dependencies within single workspaces, we need a dependency relation that is based on update-in-place policy. Therefore, we apply the backward commutativity relation bc [Wei89]. The bc relation can be informally described as follows: two activity instances a and b commute backward if, for all workspace states in which a can be executed directly after b or vice versa, the execution order can be exchanged without affecting the output values of a and b, and without affecting any subsequent activity instance. Note that the relation is symmetric but not transitive.

Definition 5 (backward commutativity relation) We introduce the *backward commutativity* relation bc which is defined over $\Sigma \times \Sigma$:

$$\begin{array}{ll} \forall a, b \in \Sigma: & (a, b) \in bc :\Leftrightarrow \\ \forall \text{ histories } h: & (1)\ \ h \bullet [a] \bullet [b] \text{ legal } \Leftrightarrow h \bullet [b] \bullet [a] \text{ legal} \\ & (2)\ \ h \bullet [a] \bullet [b] \equiv h \bullet [b] \bullet [a] \end{array}$$

The relation bc allows us to define certain properties on histories:

Definition 6 (closed subhistory) A subhistory $h' = (ACT', \ll_{ACT'})$ of history $h = (ACT, \ll_{ACT})$ is *closed* iff

$$\forall a \in ACT, a' \in ACT', (a, a') \in \ll_{ACT}: (a, a') \notin bc \Rightarrow a \in ACT'$$

A closed subhistory contains for each activity instance every preceeding activity instance it depends on. We consider a closed subhistory as a consistent unit of work.

Theorem 1 Each closed subhistory of a legal history is legal.

Proof sketch By a simple induction we can show that each activity instance that is not part of the closed subhistory can be removed with the resulting history remaining legal. The induction step is as follows: Consider the last activity instance that is not part of the closed subhistory. Then, by definition of a closed subhistory, we know that it commutes backward with all subsequent activity instances. Hence, we can move it to the end of the history and, thus, it can be omitted. □

Definition 7 (minimal closed subhistory) A closed subhistory $h' = (ACT', \ll_{ACT'})$ of $h = (ACT, \ll_{ACT})$ is *minimal* under $A \subseteq ACT$ iff

$$\forall a' \in ACT' : (a' \in A) \vee (\exists b' \in ACT' : (a', b') \in \ll_{ACT'} \wedge (a', b') \notin bc)$$

A history is minimal closed under A iff it contains only activity instances of A or those activity instances of ACT, activity instances of A (possibly transitively) depend on.

Definition 8 (independent subhistories) Two subhistories $h' = (ACT', \ll_{ACT'})$, $h'' = (ACT'', \ll_{ACT''})$ of $h = (ACT, \ll_{ACT})$ are *independent* subhistories iff

$$\begin{array}{l} ACT' \cap ACT'' = \emptyset \wedge \\ \forall a' \in ACT', a'' \in ACT'' : \quad (a', a'') \in bc \end{array}$$

Two subhistories are independent if they contain disjoint sets of activity instances and there exist no relevant dependencies between them. Two independent closed subhistories correspond to consistent units of work that can be exchanged independently.

4.6 Compensation

To allow for the interactive exploration of different problem solutions that are carried out within the private workspaces, we require recovery services that allow to retract decisions taken by a single actor (by the compensation of activity instances).

Definition 9 (compensation) The compensation function $comp$ given in the specification is defined as follows:

$$\begin{array}{l} \forall a, a^{comp} \in \Sigma : comp(a) = a^{comp} :\Leftrightarrow \\ \quad \forall \text{ history } h : h \bullet [a] \text{ is legal } \Rightarrow h \bullet [a] \bullet [a^{comp}] \text{ is legal } \wedge h \equiv h \bullet [a] \bullet [a^{comp}] \end{array}$$

The interpretation of this definition is that the execution of a^{comp} immediately after a is simply the undo of a. This means that no subsequent activity instance can observe that the sequence $[a] \bullet [a^{comp}]$ is part of the history. Note, that it is sufficient for the resulting state to be *equivalent* to the state that would have been reached if the compensated-for activity would never have been executed, i.e., the states need not be identical [KLS90].

Compensation becomes more difficult if we want to compensate an activity a in case further activity instances have been performed, yet. It may be the case that these subsequent activity instances are dependent on the results of a such that a cannot be compensated without affecting its successors.

Definition 10 (compensatable) An activity instance a of a history $h = (ACT, \ll_{ACT})$ with $comp(a) = a^{comp}$ is *compensatable* iff

$$h \bullet [a^{comp}] \text{ is legal } \wedge \forall b \in ACT : (a, b) \in \ll_{ACT} \Rightarrow (b, a^{comp}) \in bc$$

If a is compensatable in a history h we can directly apply the compensation. Otherwise, we have to compensate all activities b with $(a, b) \in \ll_{ACT}$ and $(b, a^{comp}) \notin bc$ to correctly compensate a. This ensures that compensation does not disturb the outcome of dependent activities.

4.7 Merging of histories

The bc relation introduced above allows us to identify consistent units of work that can be subject of an information exchange. If such a unit of work represented by a closed subhistory of a workspace history, is incorporated into another workspace history, we have to ensure that the exchanged activity instances behave in the destination history as in the source history.

Since we are here interested in determining dependencies between different workspaces, we need a dependency relation that is based on deferred update policy. Therefore, we apply the forward commutativity relation *fc* [Wei89]. The *fc* relation can be informally described as follows: two activity instances a and b commute forward if, for all workspace states in which a as well as b can be executed, they can be executed successively in arbitrary order without affecting the output values of a and b, and without affecting any subsequent activity instance. Note that the relation is symmetric but not transitive.

Definition 11 (forward commutativity relation) We introduce the *forward commutativity* relation *fc* which is defined over $\Sigma \times \Sigma$:

$$\begin{array}{ll} \forall a, b \in \Sigma : & (a, b) \in fc :\Leftrightarrow \\ \forall \text{ histories } h : & h \bullet [a] \text{ legal } \wedge h \bullet [b] \text{ legal } \Rightarrow \\ & (1)\, h \bullet [a] \bullet [b] \text{ legal } \wedge h \bullet [b] \bullet [a] \text{ legal} \\ & (2)\, h \bullet [a] \bullet [b] \equiv h \bullet [b] \bullet [a] \end{array}$$

We call two histories WS', WS'' of a cooperative activity CA *mergeable* iff both histories are legal. Since both WS' and WS'' belong to the same cooperative activity, they are both executions of the same cooperative activity type. Additionally, it is guaranteed that all workspaces WS_i of CA are based on the same initial workspace state S_{init}. A merged activity history $M = (ACT_M, \ll_{ACT_M})$ is constructed out of two mergeable activity histories $WS' = (ACT', \ll_{ACT'})$, and $WS'' = (ACT'', \ll_{ACT''})$. In the following, we use $WS = (ACT, \ll_{ACT})$ to refer to one of the two workspaces WS' or WS'' to avoid symmetric conditions. We use $\widehat{WS} = (\widehat{ACT}, \ll_{\widehat{ACT}})$ and $WS = (ACT, \ll_{ACT})$ to refer to different histories of $\{WS', WS''\}$.

Definition 12 (correct merged history) We call M a *correct merged* history iff

1. $ACT_M \subseteq ACT \cup \widehat{ACT}$

2. $\forall a, b \in ACT : b \in ACT_M \wedge (a, b) \in \ll_{ACT} \wedge (a, b) \notin bc \Rightarrow a \in ACT_M$

3. $\forall a, b \in ACT : (a, b) \in \ll_{ACT} \wedge (a, b) \notin bc \wedge a, b \in ACT_M \Rightarrow (a, b) \in \ll_{ACT_M}$

4. $\forall a \in (ACT \setminus \widehat{ACT}), \forall a' \in (\widehat{ACT} \setminus ACT) : (a, a') \notin fc \Rightarrow a \notin ACT_M \vee a' \notin ACT_M$

First, we state that the merged history can only be constructed out of activity instances from the two input histories WS' and WS''. Second, it is expressed that all relevant predecessor activity instances of any activity instance that is part of the merged history have to be included in the merge, too. Third, we require that a relevant ordering of activity instances in a source history is preserved in the merged history. Fourth, if two activity instances a and a' not contained in both histories with a not forward commuting with a', only one of them can be part of the merged history.

Theorem 2 A correct merged history is legal.

Proof sketch From properties (2) and (3) we can deduce that the subhistory of M consisting of activity instances of ACT is equivalent to a closed subhistory of WS in the sense that it differs only in the order of activity instances that commute backward. Hence, by theorem 1 it is legal. Property (4) guarantees that the history remains legal if activity instances from both histories interleave. The fact that property (1) ensures that there cannot be any further activity instances finally provides us with the theorem. □

4.8 Merge Algorithm

In the following we describe the merging of activity instances from workspace history $WS' = (ACT', \ll_{ACT'})$ into workspace history $WS = (ACT, \ll_{ACT})$[1]. For illustrative examples of the merge concept we refer the reader to [RKT+95, WK96]. The merge process is performed in five steps:

1. Select activity instances from the source history WS'
2. Compute closed subhistory (exchange history)
3. Partition the exchange history into the maximal number of independent closed subhistories
4. Select activity instances from partitions along their dependencies
5. Merge the selected activity instances into destination history

As a first step, the controlling user selects a set $I \subseteq ACT'$ of activities to be merged into WS. The activities in I do not necessarily comprise a consistent unit of work. To satisfy property (2) of correct merged activity histories, we construct the minimal closed subhistory $P = (ACT_P, \ll_P)$

[1]Note, that the resulting history of the merge process as well as all intermediate histories that are constructed during the merge process are also denoted by WS.

under I. The subhistory P is the part of WS' to be merged into WS, thus, we call it the *exchange history*.

During the merge, there may occur conflicts between activity instances of WS and P. If we allow only an atomic merge of P into WS, i.e., to include all or none activity instances of P, the mechanism would be rather inflexible. In case of a conflict we either have to discard a large portion of WS or we cannot include any activity instance of P at all. Therefore, to minimize the loss of work during the merge process we allow for a partial merge, i.e., to include only parts of P into WS. Hence, we partition P into independent closed subhistories $(P_1, \ldots, P_k)$. Each $P_i = (ACT_i, \ll_{P_i})$ can be merged separately into WS. Note, that each ACT_i contains at least one $a \in I$.

The fact that the relations $bc^c \cap \ll_{P_i}$ are asymmetric[2] can be used to achieve an even finer merge granularity. Hence, we can merge each subset of ACT_i corresponding to a closed subhistory of P_i. Thus, we can include $a \in ACT_i$ into the destination history WS if all other members of the minimal closed subhistory of P_i under $\{a\}$ are already included in WS. To illustrate our approach, we introduce the $\prec$ relation:

Definition 13 We define the relation $\prec_{P_i}$ for each P_i as follows:

$$\forall a, b \in ACT_i : (a, b) \in \prec_{P_i} :\Leftrightarrow (a, b) \in \ll_{P_i} \wedge (a, b) \notin bc$$

Given a partition P_i with its corresponding relation $\prec_{P_i}$, we start merging with an element a of history P_i that does not depend on any other preceeding activity instance in P_i, i.e., $\forall b \in ACT_i : (b, a) \notin \prec_{P_i}$. Afterwards, we can merge activity instances from history P_i that are only dependent on activity instances already included in the merged history WS.

Formally, to fulfil property (2) and (3) when merging an activity instance a from P_i into WS, we require:

Condition 1

$$\forall b \in ACT_i : (b, a) \in \prec_{P_i} \Rightarrow b \in ACT$$

To check whether the inclusion of $a \in ACT'$ into history $WS = (ACT, \ll_{ACT})$ is in line with property (4) of correct merged histories, we require:

Condition 2

$$\forall b \in ACT \setminus ACT' : (a, b) \in \mathrm{f}c$$

This condition ensures that activity instance a can only be appended to WS if it commutes forward with all activities in ACT that are not in ACT'. Since we deduced conditions 1 and 2 straightforward from definition 12, the algorithm obviously constructs a correct merged history.

If condition 2 is not fulfiled there are two possibilities how to proceed. First, the user can decide to abandon the inclusion of a into the merged history. This implies that we also have to abandon the inclusion of all remaining activity instances a' in P_i that depend on a, i.e., $(a, a') \in \prec_{P_i}$. Second, the activities in ACT that are not forward commuting with a can be compensated. In this case it is possible to include further activity instances from P_i.

When all partitions P_i are processed by the merger, the algorithm terminates. Note, that this does not imply that all $a \in I$ have been incorporated into WS since the controlling user (or the software module) might have decided during the merge process to leave out some activity instances of I.

[2] With bc^c we denote the complement of bc.

4.9 Semantic relations beyond commutativity in the context of CoAct

In the previous sections we argued for the use of backward and forward commutativity to determine dependencies between activity instances because they are based on update-in-place policy and deferred update policy, respectively. Commutativity relations are obviously not the only ones that fulfil these requirements.

Commutativity relations pose two requirements on a pair of activity instances. First, they require that the execution is legal in both orders and second, they demand that the states resulting from both execution orders are equivalent. A natural modification to reduce the number of conflicts is to drop the requirement for equivalent states (as proposed in [Wäs96]).

In the following we therefore examine two other relations, namely *recoverability* (*rcv*) [BR92] and *ignores* (*ign*) [Her90][3], and discuss their applicability in the context of our model.

Definition 14 (recoverability relation)

$$\forall a, b \in \Sigma : (b, a) \in rcv \quad :\Leftrightarrow \forall \text{ history } h_1, h_2 : \quad \begin{array}{l} h_1 \bullet h_2 \text{ is legal } \wedge \\ h_1 \bullet [a] \bullet h_2 \bullet [b] \text{ is legal } \Rightarrow \\ h_1 \bullet h_2 \bullet [b] \text{ is legal} \end{array}$$

Definition 15 (ignores relation)

$$\forall a, b \in \Sigma : (b, a) \in ign \quad :\Leftrightarrow \forall \text{ histories } h_1, h_2 : \quad \begin{array}{l} h_1 \bullet [a] \bullet h_2 \text{ is legal } \wedge \\ h_1 \bullet h_2 \bullet [b] \text{ is legal } \Rightarrow \\ h_1 \bullet [a] \bullet h_2 \bullet [b] \text{ is legal} \end{array}$$

These relations do not consider effects on subsequent activity instances. In case of recoverability it is considered whether an activity instance is independent of a previous activity instance, i.e., the previous activity instance is not needed to guarantee a legal execution. In case of ignores an activity instance is not invalidated if another activity instance is executed previously, i.e., it remains legal.

Since rcv considers the removement of activity instances from histories and ign their insertion, the rcv relation may be used to compute closed subhistories (replace bc) and the ign relation may be used to determine conflicts between different workspaces (replace *fc*).

Note, that these relations are not weaker than backward and forward commutativity, respectively, i.e., $bc \not\subseteq rcv$ and $fc \not\subseteq ign$. This results from the fact that commutativity relations, in contrast to rcv and ign, examine only the execution of activity instances that follow directly one after the other. For an illustrative example of this difference, we refer to [Her90].

The lack of assertions about states has some interesting consequences. For instance, two activity instances, each consisting of a "blind write", are recoverable and ignore each other although they neither commute forward nor backward. This implies that, e.g., a single "blind write" corresponds to a consistent unit of work and can be subject of an information exchange. For the same reason, there are no conflicts if it is imported into the destination workspace.

We now examine the applicability of these relations and demonstrate that even for rather simple examples severe problems occur. Let us consider the case where both commutativity relations are replaced by the relations described above. We denote activity instances as introduced

[3]Strictly speaking Herlihy defines the complement of ign, called *invalidated-by*

rcv	(write, $\{x\}, \emptyset$)	(read, $\emptyset, \{x\}$)
(write, $\{y\}, \emptyset$)	true	true
(read, $\emptyset, \{y\}$)	$x \neq y$	true

ign	(write, $\{x\}, \emptyset$)	(read, $\emptyset, \{x\}$)
(write, $\{y\}, \emptyset$)	true	true
(read, $\emptyset, \{y\}$)	$x = y$	true

Table 1: Recoverability and ignores relations for read/write

in section 4.2, e.g., $(2, \text{write}, \{7\}, \emptyset)$ represents an instance of the activity type 'write' with the value to be written 7, no output, and the identifier 2. To specify conflict relations we use the same notation without the leading identifier. In table 1, we present the relations for reading and writing the value of a single data item. An entry indicates that the row is independent on the column when the indicated condition holds.

Let us assume that a user has executed in his workspace the following sequence of activity instances:

$$WS_1 : (1, \text{write}, \{5\}, \emptyset), (2, \text{write}, \{7\}, \emptyset), (3, \text{read}, \emptyset, \{7\})$$

Suppose, another user wants to import activity instance 2. The closed subhistory of $(2, \text{write}, \{7\}, \emptyset)$ is the activity instance itself since, according to the definition of rcv, a write never depends on another activity instance. The destination history is empty, thus, we do not encounter any conflicts. If the second user next decides to import also activity instance 1, he is allowed to do so, and we get the following situation:

$$WS_1 : (1, \text{write}, \{5\}, \emptyset), (2, \text{write}, \{7\}, \emptyset), (3, \text{read}, \emptyset, \{7\})$$
$$WS_2 : (2, \text{write}, \{7\}, \emptyset), (1, \text{write}, \{5\}, \emptyset)$$

If the second user finally decides to import activity instance 3 he would also be allowed to do so because the two conditions of our merge algorithm are satisfied:

(1) The only activity instance in the source history activity instance 3 depends on, namely $(2, \text{write}, \{7\}, \emptyset)$, is already included in WS_2. Hence, condition 1 is satisfied.

(2) All activity instances of WS_2 are already contained in WS_1. Hence, condition 2 is satisfied. Note, that we use here our notion of *identical* activity instances.

Nevertheless, the resulting history would not be legal. On the other hand, if we had used backward commutativity to determine closed subhistories, the first exchange would have resulted in the transfer of both instances of activity type 'write' from WS_1 to WS_2, thereby ensuring that they occur in the appropriate order in WS_2.

One may argue that the reason for this anomaly is not the usage of relations that do not guarantee the equivalence of states but rather the policy of our exchange algorithm to check only for conflicts between activity instances that have not yet occured in the source workspace

rcv	(deposit, $\{x\}, \emptyset$)	(withdraw, $\{x\}, \{ok\}$)
(deposit, $\{y\}, \emptyset$)	true	true
(withdraw, $\{y\}, \{ok\}$)	false	true

ign	(deposit, $\{x\}, \emptyset$)	(withdraw, $\{x\}, \{ok\}$)
(deposit, $\{y\}, \emptyset$)	true	true
(withdraw, $\{y\}, \{ok\}$)	true	false

bc	(deposit, $\{x\}, \emptyset$)	(withdraw, $\{x\}, \{ok\}$)
(deposit, $\{y\}, \emptyset$)	true	false
(withdraw, $\{y\}, \{ok\}$)	false	true

Table 2: Recoverability, ignores and backward commutativity relations for a bank account

(condition 2). Our next example shows the virtue of this policy, and that its omission would introduce different anomalies.

Consider the well known example of a bank account where users can deposit and withdraw money with the restriction that the balance must not become negative [Wei88, Her90]. Then we have dependency relations as in table 2. Now, lets have a look at the following history:

$$WS_1 : (1, \text{deposit}, \{100\}, \emptyset), (2, \text{withdraw}, \{40\}, \{ok\}), (3, \text{withdraw}, \{50\}, \{ok\})$$

Assume, another user simply wants to import the whole history. In case the merge algorithm checks conflicts between activity instances that have already been executed in the source workspace, this would not be permitted. After activity instances 1 and 2 are imported, an attempt to import the third results in a conflict.

Even if we omit the conflict checks between activity instances that have already been executed in one workspace in the same order would be too strict. In this case we would not be able to import the second instance of type withdraw before the first one, i.e., we would not allow

$$WS_2 : (1, \text{deposit}, \{100\}, \emptyset), (3, \text{withdraw}, \{50\}, \{ok\}), (2, \text{withdraw}, \{40\}, \{ok\})$$

This sequence would on the other hand be allowed if we neglect conflicts between activity instances that have already been executed in the source workspace regardless of the type of relations we use.

To summarize our comparison we state that the use of commutativity relations has the advantage that they (by nature of commutativity) exactly tell us when we are allowed to merge operations in a different order than originally executed (like in the second example) and when we are not allowed to do so, i.e., better import some previous activity instances, too (like in the first example).

Similar problems as described above occur if we replace only a single commutativity relation.

5 Conclusion and related work

In this paper, we have presented a formal description of the CoAct model [RKT+95, WK96]. CoAct provides a flexible framework to support cooperation in interactive, multi-user environments. After having given an overview of CoAct we described how we model the execution of a single user's work by means of workspace histories. We presented the merge mechanism that permits the semantically correct exchange of information among concurrent activities of cooperating users and showed that correct merged histories remain legal. Furthermore, we discussed an algorithm which utilizes the idea of history merging to implement CoAct's exchange operations (import, delegate, and save).

As pointed out in the introduction, checkout models [BKK85, KSUW85] do not explicitly address cooperation among co-workers — although if they appear in tandem with versions and configurations. Checked-out objects are reserved for exclusive access until a later check-in. Another drawback is that all objects needed to perform a specific task have to be checked out explicitly. For software development applications, several extensions of the basic checkout model have been developed by taking advantage of the opportunity of generic software consistency checking [Hon88, KPS89], but they are not applicable in general.

To ensure the correctness of cooperation in CoAct, we exploit the type-specific semantics of activity instances by means of *both* backward and forward commutativity relations. We use backward commutativity to identify dependencies between activity instances of the same workspace (history). We use forward commutativity to identify conflicts among activity instances executed in different workspaces.

Regarding the units of exchangeable activity instances we follow a similar approach as 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. We use the dependencies between activity instances (induced by backward commutativity) to identify consistent units of work (closed subhistories) that can be subject to an exchange of activity instances among workspaces.

The concept of delegation can also be found in ACTA [CR92]. 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 [CR93, CR94]. How delegation in ACTA can be implemented is described in [MR95]. In our model 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 split/join transaction model [PKH88, KP92] supports the dynamic restructuring of ongoing transactions. A split-operation allows to split a running transaction into two new (serial or independent) transactions. A join-operations allows to incorporate two transactions into a new transaction. These operations are based on the consideration of the read- and write-sets of flat transactions. Higher level operation semantics are not taken into account. The split/join transaction model can be extended in order to be more powerful and flexible in splitting and joining, i.e., merging, ongoing transactions by utilizing the semantics of operations and applying our merge criteria, thus, enabling a closer cooperation between transactions.

The participant transaction model [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 to be serializable (only

transactions of different domains have to be serializable). Because 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 unintuitive behaviour of the system from the user's viewpoint. In CoAct we guarantee that the view of each activity instance remains valid by exploiting its semantics. The system detects if the exchange of activity instances would introduce inconsistencies and proposes alternative sets of activity instances that guarantee consistency of data.

In section 4.9, we introduced semantic relations that rely only on the values that are externalized to a user (and not on the resulting workspace states). We discussed the impact of these relations, namely recoverability and ignores, on CoAct's merge mechanism. We showed that commutativity behaves more natural because the above relations suffer from certain anomalies (discussed in section 4.9) that may occur during merging histories even for rather simple examples. For this reason, we think that the usage of backward and forward commutativity is more intuitive from a user's viewpoint.

The recoverability relation we used in this paper is based on [BR92]. The property of recoverability is utilized in [BR92] to decrease the delay involved in processing non-commuting operations of different transactions while still avoiding cascading aborts. In this paper, we examined the usage of recoverability to identify consistent units of work within a single workspace history.

The ignores relation we introduced is based on the invalidated-by relation [Her90]. Invalidated-by is a serial dependency relation that is used in the context of optimistic concurrency control to define legal histories and serializability of transactions. In this paper, we investigated if ignores can be utilized to reduce the amount of actions that have to be discarded during a merge.

Compared to semantics-based optimistic concurrency control we do not assume that different histories are disjoint which is an implicit assumption in [Her90]. We use the concept of identical activity instances to establish a closer cooperation among co-workers and their workspace (histories).

In CoAct, we achieve cooperation in terms of exchanging activity instances (and not data items) among workspaces. One may argue that the exchange of results in terms of data or documents, respectively is more natural from a user's viewpoint. We want to point out that the exchange of data items can be mapped to the exchange of a respective set of activity instances contained in a workspace history. Moreover, using of activity instances and exploiting their semantics enables a higher degree of cooperation than only exchanging data items (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 run-time, 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.

In the context of the TransCoop project we are currently implementing the basic building blocks of the CoAct model (workspaces, histories, merge mechanism) within the object-oriented DBMS VODAK [GI95]. VODAK already applies user-specified commutativity relations for concurrency control. The application scenario for the TransCoop demonstrator system is the hypermedia authoring system SEPIA [WA95] which will be enriched by the CoAct cooperation facilties of the VODAK database system.

Acknowledgments

The authors would like to thank all members of the TRANSCOOP teams in Darmstadt, Enschede, and Helsinki for many inspiring discussions on the topic.

References

[BBG+83] C. Beeri, P. A Bernstein, N. Goodman, M. Y. Lai, and D. E. Shasha. A concurrency control theory for nested transaction. In *Proc. of the second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1983.

[BE96] O. A. Bukhres and A. K. Elmagarmid, editors. *Object-oriented Multidatabase Systems*. Prentice-Hall, 1996.

[BKK85] F. Bancilhon, W. Kim, and H. Korth. A model of CAD transactions. In *Proc. of the 11th Int. Conference on Very Large Databases*, pages 25–33, Stockholm, Sweden, August 1985.

[BR92] B. R. Badrinath and K. Ramamritham. Semantic–based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[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.

[EGR91] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[Elm92] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. ACM Press. Morgan Kaufmann Publishers, Inc., 1992.

[FÖ89] A. A. Farrang and M. T. Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[GI95] GMD-IPSI. VODAK V4.0 User Manual. Arbeitspapiere der GMD 910, Technical Report, GMD, April 1995.

[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.

[Hon88] M. Honda. Support for parallel development in the Sun network software environment. In *Proc. of the second Int. Workshop on Computer-Aided Software Engineering*, pages 5–5 – 5–7, Cambridge, Massachusetts, USA, July 1988.

[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.

[Kai95] G. E. Kaiser. Cooperative transactions for multiuser environments. In Kim [Kim95], chapter 20, pages 409–433.

[Kim95] W. Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and beyond*. Addison-Wesley Publishing Company, 1995.

[KL89] W. Kim and F. H. Lochovsky, editors. *Object–Oriented Concepts, Databases, and Applications*. ACM Press, 1989.

[KLS90] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating tranactions. In *Proc. of the 16th Int. Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, 1983.

[Kor95] H. F. Korth. The double life of the transaction abstraction: Fundamental principle and evolving system concept. In *Proc. of the 21st Int. Conference on Very Large Databases*, pages 2–6, September 1995. Zurich, Switzerland.

[KP92] G. E. Kaiser and C. Pu. Dynamic restructuring of transactions. In Elmagarmid [Elm92], chapter 8, pages 265–295.

[KPS89] G. E. Kaiser, D. E. Perry, and W. M. Schell. Infuse: Fusing integration test management with change management. In *Proc. of the 13th IEEE Computer Software and Applications Conference*, pages 552–558, Orlando, Florida, USA, September 1989.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 388–401, Austin, Texas, USA, May 1985.

[LMWF94] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., 1994.

[MR95] C.P. Martin and K. Ramamritham. Delegation: Efficiently rewriting history. Technical Report 95-90, Dept. of Computer Science, University of Massachusetts, October 1995.

[MRW+93] P. Muth, T. C. Rakow, G. Weikum, P. Brössler, and C. Hasse. Semantic concurrency control in object–oriented database systems. In *Proc. of the 9th IEEE Int. Conference on Data Engineering*, pages 233–242, 1993. Vienna, Austria, April 19–20.

[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.

[PKH88] C. Pu, G. E. Kaiser, and N. Hutchinson. Split–transactions for open–ended activities. In *Proc. of the 14th Int. Conference on Very Large Databases*, pages 26–37, Los Angeles, California, USA, August 1988.

[RKT+95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a cooperative transaction model: The cooperative activity model. In *Proc. of the 21st Int. Conference on Very Large Databases*, pages 194–205, September 1995. Zurich, Switzerland.

[SS84] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[SZ89] A. H. Skarra and S. B. Zdonik. Concurrency control and object-oriented databases. In Kim and Lochovsky [KL89], chapter 16, pages 395–421.

[TV95] T. Tesch and P. Verkoulen. Deliverable II.2: requirements for the TransCoop transaction model. Report TC/REP/GMD/D2-2/207, Esprit Project No. 8012, 1995.

[TW95] T. Tesch and J. Wäsch. Transaction support for cooperative hypermedia document authoring: A study on requirements. In *Proc. of 8th ERCIM Database Research Group Workshop on Database Issues and Infrastructure in Cooperative Information Systems*, pages 31–42, Trondheim, Norway, August 1995.

[VFSE95] P. A. C. Verkoulen, F. J. Faase, A. W. Selders, and P. J. J. 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. Begell House, Inc. New York - Wallingtord (U.K.), June 1995. Stuttgart, Germany.

[VRS86] S. Vinter, K. Ramamritham, and D. Stemple. Recoverable actions in Gutenberg. In *Proc. of the sixth IEEE Int. Conference on Distributed Computing Systems*, pages 242–249, May 1986. Washington, D.C.

[WA95] J. Wäsch and K. Aberer. Flexible design and efficient implementation of a hypermedia document database system by tailoring semantic relationships. In *Proc. of the sixth IFIP Conference on Database Semantics*, May 30 – June 2 1995. Atlanta, Georgia.

[Wäs96] J. Wäsch. History merging as a mechanism for information exchange in cooperative and mobile environments. *Datenbank-Rundbrief*, 17, May 1996.

[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.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[Wei89] W. E. Weihl. The impact of recovery on concurrency control. In *Proc. of the 8th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 259–269, 1989.

[WK96] J. Wäsch and W. Klas. History merging as a mechanism for concurrency control in cooperative environments. In *Proceedings of RIDE-Interoperability of Nontraditional Database Systems*, pages 76–85, New Orleans, USA, February 1996.

[WL93] U. K. Wiil and J. J. Leggett. Concurrency control in collaborative hypertext systems. In *Proc. of the fifth ACM Conference on Hypertext*, pages 14–18, November 1993. Seattle, Washington.

[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.