Towards a Cooperative Transaction Model – The Cooperative Activity Model –

M. Rusinkiewicz
University of Houston
Dept. of Computer Science
Houston, TX 77204
USA
marek@cs.uh.edu

W. Klas T. Tesch J. Wäsch
GMD-IPSI Integrated Publication and Information Systems Institute
D-64293 Darmstadt
Germany
{klas,tesch,waesch}@darmstadt.gmd.de

P. Muth
University of the Saarland
Dept. of Computer Science
D-66041 Saarbrücken
Germany
muth@cs.uni-sb.de

Abstract

With the emergence of cooperative applications it turned out that traditional transaction concepts are not suitable for these scenarios. Isolation of transactions, as guaranteed by the ACID transaction properties, contradicts the need of cooperation between users. In this paper, we propose a cooperative activity model that provides transactional properties suitable for cooperative scenarios. Each user participating in a cooperative activity has his own private workspace that is isolated from other users. Cooperation is achieved by controlled exchange and synchronization of the contents of workspaces among the users and by installing results of their activities in the common activity database. Our model ensures that the joint execution of a cooperative activity is equivalent to one that could have been carried out by a single user. We discuss our cooperative activity model in different scenarios, one requiring a close cooperation in an authoring environment, and the second implementing a workflow-like scenario.

1 Introduction

With the emergence of new application areas, the basic concept of transaction is being re-examined and new extended transaction models are being proposed. The traditional transaction model was based on the assumption that a large number of relatively short-lived transactions are accessing a shared database. This led to the emergence of the ACID model [HR83] that is based on isolation among transactions and uses serializability as a natural correctness criterion for the concurrent execution of transactions.

The new application areas in which the transaction concepts are being applied, such as CAD, CASE, automated office workflows, or cooperative authoring, have quite different characteristics. The transactions in such environments need to support long-running activities, in which competition for resources (on which the locking protocols were 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.

At the same time, we are interested in preserving some transactional properties of activities performed in these cooperative environments. With the traditional transactions, the user (transaction programmer and the end-user) is freed from concerns about partial (and therefore incorrect) results of transactions being left in the database in the case of a failure, or being observed by other users, concurrently accessing the system. These concerns are addressed by the transaction processing system that enforces commitment protocols ensuring transaction failure atomicity and concurrency control ensuring that the effect of concurrent execution of transactions is equivalent to that of running these transactions one at a time, without any interference from each other.

We argued that in a cooperative environment these properties may be not useful. Failure atomicity may be too strict and isolation among concurrent users may be undesirable. However, we need to replace them by new criteria, more suitable for cooperative activities. In this paper, we present a proposal of the *Cooperative Activity Model* (CoAct) we developed in the context of TransCoop[1]. Instead of atomicity,

**Proceedings of the 21th VLDB Conference
Zurich, Switzerland, 1995**

[1]This work is partially done in the ESPRIT LTR 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).

our model guarantees that cooperative activities are executed in accordance with *execution rules* defined by the designer of a given activity and *terminate* (commit) only in legal *termination states*. Instead of isolation, we guarantee that although the information is exchanged between the users concurrently participating in a cooperative activity, the result of such concurrent execution is equivalent to that of executing this activity by a single fictitious user having all the knowledge and authorizations of the cooperating users.

The paper is organized a follows. In section 2, we discuss the basic components of a cooperative activity type specification. In section 3, the implementation of a cooperative activity by participating user activities is explained. In section 4, we review the related work in the area of cooperative transactions and workflows and compare it to our approach. Finally, we present conclusions and further work.

2 Specification of Cooperative Activities

2.1 A Framework for Cooperative Activities

In the CoAct model *cooperative activities* are specified by means of parameterized *cooperative activity types*. A cooperative activity type describes identical activities occuring in a particular organization. Each cooperative activity is described by a single cooperative activity type (*CAT*). A concrete cooperative activity (*CA*) is an instantiation of a cooperative activity type.

Usually, more than one user participates in a cooperative activity. We assign a *user activity* to each user participating in a cooperative activity. A user activity (*UA*) belongs to exactly one cooperative activity. The user activities are executed interactively by the responsible users in accordance with the pre-defined execution constraints of the cooperative activity type. Therefore, the exact sequence of operations performed within user activities cannot be determined beforehand.

Each user executing a user activity has his own *private workspace* that is isolated from the workspaces of the other users. Additionally, there exists a common activity database, shared by all users, that stores common data.

Similar to types of cooperative activities, users are described and classified by specific *user types* depending on the role they play in the organization and the cooperative activity, respectively. The allowed interactions between the users participating in a cooperative activity may depend on this classification. We will not discuss this issue further, because user modelling is beyond the scope of this paper.

Cooperation between user activities belonging to the same cooperative activity is achieved by controlled exchange of (parts of) the contents of private workspaces. This mechanism can also be interpreted as synchronization of (parts of) user activities. Another way users can collaborate is by installing possibly partial, but consistent results of their user activities in the common activity database. Other users can access the current state of the cooperative activity kept in the common database and import (parts of) this cooperative activity into their private workspaces.

The model guarantees that, although information is exchanged between user activities concurrently executing a cooperative activity, the result of a concurrent execution is equivalent to executing the cooperative activity in a single thread of control. Traditional concurrency control assumes that transactions are correct and guarantees that no errors will be introduced by their interleaved execution.

In our model, we assume that user activities are correct (obey all execution rules) and we guarantee that no errors will be introduced by the merging of concurrently executed threads. Although various (partial) versions of the activity may exist in private workspaces at different times, there is a single final result of a cooperative activity. This is achieved by *merging* the results of individual user activities of a cooperative activity in a semantically consistent way. Merging of user activities is described in section 3.2 in more detail.

To support the exchange of data and the merging process and to ensure that users participating in the same cooperative activity are aware of each other and of the progress of the cooperative activity, additional *coordination facilities* are provided. For example, a user can be notified if another user joins or leaves the cooperative activity or if someone wants to exchange information with a co-worker.

Given this framework the concepts of the CoAct model can be summarized as follows:

- *cooperative activity types* describing cooperative activities,
- *cooperative activities* as concrete instantiations of cooperative activity types and as a composition of individual user activities,
- *user activities* as activities of individual participating users collaborating in a cooperative activity,
- a *common activity database* for a cooperative activity,
- *private workspaces* for individual user activities,
- a *merger* supporting the correct exchange of information between collaborating users, and
- *coordination facilities* providing awareness and coordination of users.

Figure 1 gives a simple illustration of the concepts discussed so far. The cooperative activity types X and

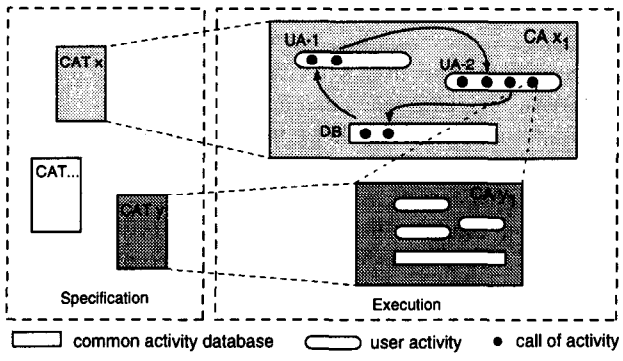

Figure 1: Concepts of CoAct

Y are instantiated in the execution environment. Each cooperative activity (X_1 and Y_1) consists of individual user activities and a common activity database. The call of another cooperative activity is presented within user activity *UA-2*. The arrows illustrate the different options for the exchange of information (import / save / delegate) within the scope of a cooperative activity.

2.2 Cooperative Activity Types

A cooperative activity type consists of the following five components:

1. Activity declaration

The declaration of a cooperative activity type starts with the keyword CAT, followed by a unique name N of the cooperative activity type, a definition of activity parameters P and a compensation activity C.

The definition of activity parameters includes the formal input parameters $I_1, \ldots, I_n$ and the output parameter list $O_1, \ldots, O_m$. The input and output parameters allow for a parameterized instantiation of a cooperative activity type. A declaration of a compensation activity C specifies the way the results of a cooperative activity can be semantically undone after its commitment.

2. Constituent subactivities

Constituent subactivities are specified as a set $\mathcal{A}$ of activities. A constituent subactivity may be another cooperative activity specified by a cooperative activity type or it may be an *elementary activity*. An elementary activity, described by an elementary activity type *EAT*, is a regular transaction program in the host OODBMS which consists of a (sequence of) method invocation(s) or a call to an external program. The ability to incorporate external applications via declaration as elementary activity types is particularly important if a cooperative activity is used to model the execution of a workflow — a multi-step activity involving coordinated execution of tasks running on different systems [RB94].

Each subactivity is assigned to a unique identifier A_i with i identifying a particular subactivity. A subactivity can be associated (explicitly or implicitly through its type definition) with a user type to restrict the execution permission to responsible users. Furthermore, we specify for each subactivity how many times it can occur in the final history of the cooperative activity. We write

$$\{A_i\}^n :: \text{ACTIVITYTYPE}(In : \ldots, Out : \ldots) by \text{ USERTYPE}$$

to denote that subactivity A_i can occur n times while simply one occurrence is denoted by "$A_i :: \ldots$". It should be noted, that the used input and output parameters may introduce an implicit order between the constituent subactivities (data-flow).

3. States and state transitions of the activity

States of the cooperative activity and their state transitions are described as a set of *observable states* $\mathcal{S}$ and possible transitions $\mathcal{ST} : \mathcal{S} \longrightarrow \mathcal{S}$. Transitions between the states of the cooperative activity can be represented by a finite state automaton and its transition graph [ARSS93].

When composing a cooperative activity type and defining its execution constraints, we are only interested in those states of constituent subactivities that can be externally observed and only those transitions that can be externally controlled. These characteristics must be provided as a part of the definition of a cooperative activity.

Observable states are described in terms of predefined states defined for each activity which include at least the states: *not executed (NE)*, *executing (E)*, *done (D)*, and *undone (UD)*. Additional observable states can be defined as compositions of observable states and output values of constituent subactivities, or external events in terms of restricted first-order logic. For example, a "successful termination" state could only be reached if some constituent subactivities were completed successfully and before, e.g., Dec 31, 1995. The transitions defined on predefined states are shown in figure 2. In the following, we write $s(A)$ to denote that activity A is in state s.

A set of *breakpoints* $\mathcal{B} \subseteq \mathcal{S}$ defines states where the execution of a cooperative activity can be interrupted. Once a breakpoint is reached, the internal consistency constraints are satisfied, which corresponds to reaching an intermediate, but semantically consistent state in the execution of the activity. The work accomplished so far can be suspended, so that the activity can be resumed later (thus achieving the forward-recoverability postulate of ConTracts [WR92]). Alternatively, the results can be either exchanged with another user concurrently executing the same cooperative activity, or saved in the common activity database.

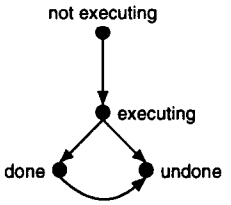

Figure 2: Transitions on predefined states

The set of breakpoints $\mathcal{B}$ contains at least one state in which the cooperative activity can terminate. *Termination states* $\mathcal{T} \subseteq \mathcal{B}$ are observable. A termination state of a cooperative activity corresponds to semantically successful or unsuccessful completion.

4. Execution rules

A set of *execution rules* $\mathcal{E}$ is specified to govern the execution of the constituent subactivities. These rules can be interpreted as integrity constraints on the occurrence and temporal precedence of observable states associated with the execution of the subactivities [RS95]. If the rules are completely defined, they correspond to a script for the execution of subactivities implementing a cooperative activity. In a correct CAT specification, we require the execution rules to be in line with the data-flow between constituent subactivities. The execution rules consist of two parts: forward execution and backward execution rules.

Forward execution rules govern the regular execution of a cooperative activity and, thus, of each involved user activity. Backward execution rules govern the rollback of the activity, if it is not finished yet and needs to be aborted. For the sake of simplicity, we assume in this paper that rollback is performed using compensation activities in inverse order. In general, execution rules can be defined in terms of:

- *Observable states* of constituent subactivities, e.g., "subactivity A_1 cannot start until subactivity A_2 is finished". For the sake of simplicity, we use predefined states only.
- *Output values* of constituent subactivities, e.g., "subactivity A_1 can start if subactivity A_2 returns an output value greater than 25".
- *External variables*, that are modified by external events that are not part of the cooperative activity, e.g., "subactivity A_1 cannot be started before 9AM".

Execution rules have been discussed among others in [SANR92, ANRS92, CR91, GHKM94]. We use the well known ordering dependency ($<$) and existence dependency ($\rightarrow$) as described in [Kle91] to express execution rules such as those in [ANRS92], e.g., a dependency A *enables* B is equivalent to the following set of dependencies: $\{Done(A) < Executing(B),\ Executing(B) \rightarrow Done(A)\}$.

The execution rules allow us to reason about the correctness of the specification of the activity, before it is executed. Of course, unenforceable things can be specified. We are particularly interested in specifications that are runtime enforceable [ARSS93]. To enforce the execution rules within user activities at runtime, we use an activity scheduler — a software module that governs the execution of user activities. It receives requests from users to perform actions, e.g., start a new subactivity, and responds by allowing those requests that are in accordance with the specification.

5. Activity interleaving rules

Activity interleaving rules $\mathcal{I}$ define a possible existence of dependencies between constituent subactivities of an activity in terms of data-flow. If two subactivities depend on each other, e.g., editing a document based on annotations previously made, this will later be relevant for a merge of the according activity with other activities. If subactivity B depends on subactivity A, and B is desired to be part of a merge, A has to be present in the merge, too. Note that such dependencies are not always given by the execution rules. Assume, for two subactivities A and B belonging to the same activity, no execution rule is defined, i.e., they can be executed any time. Even though, if A is executed before B, B might become dependent on A. This case is treated by activity interleaving rules.

In addition, for each pair of subactivities executed concurrently within different user activities, activity interleaving rules define if both of them are allowed to be present in the merge, or one of them has to be discarded. Basically, we have to consider three cases:

1. The subactivities are completely independent of each other, and both can become part of the merge.
2. The subactivities are not independent, e.g., they both update the same data in a different way. In the merge, only one of these subactivities is allowed to be present.
3. The subactivities are not independent, but because of their specific semantics, it is allowed to have both subactivities present in the merge.

Case (3) is in particular important for modelling cooperation between users. Even if they execute subactivities which are not independent, both can be reflected in the merge as discussed in the above example.

Activity interleaving rules are defined in terms of a compatibility predicate $\oplus : \mathcal{A} \times \mathcal{A} \rightarrow \{True, False\}$ between pairs of subactivities of a cooperative activity

type. The evaluation of the predicate might depend on the system state and parameter values of the given subactivities. In addition, it is sometimes useful to dynamically change the compatibility predicate, e.g., two subactivities A and B are compatible until Dec. 31, 1995 only. For the sake of simplicity, we assume the compatibility predicate to be symmetric.

If two subactivities are defined to be compatible their execution order is not relevant. Usually, subactivities are compatible if they commute. But dependent on the application semantics, non-commuting subactivities might also be compatible.

In our model, compatibility is used to determine dependencies between subactivities executed by a single user, and to decide whether two executions of the same subactivity by different users can both be present in the merge. This is allowed only if they are compatible.

In the following, we discuss a small scenario to illustrate the concept of activity interleaving rules. Assume, two users are concurrently editing a document. Available subactivities are *edit(chapter)*, *annotate(chapter)* to add an annotation to *chapter*, and *removeAnnotation(chapter)* for the removal of an annotation from *chapter*. No execution rules are defined while the following activity interleaving rules between *edit* and *annotate* are specified:

$$\begin{aligned} \oplus(edit(ch1), edit(ch2)) &= (ch1 \neq ch2) \\ \oplus(annotate(ch1), annotate(ch2)) &= True \\ \oplus(edit(ch1), annotate(ch2)) &= True \end{aligned}$$

Assume, both users start with the same version of the document and both execute *edit(ch1)*. If these user activities are merged, both subactivities are found to be incompatible according to the activity interleaving rules. One of them has to be discarded. As an alternative, assume user 1 has performed *edit(ch1)* and user 2 has performed *edit(ch2)*. These subactivities are compatible and can both be present in a merge. Assume now, both users import the new versions of the chapters in their user activities and start annotating. Each user annotates both chapters *ch1* and *ch2*. Although these subactivities update the same data (at least there is a counter representing the number of annotations), they can all be incorporated in a merge as they are compatible.

2.3 Example of a Specification of a Cooperative Activity Type

As an example, let us consider a cooperative activity of designing a telephone circuit between two points as described in [ANRS92]. We specify the cooperative activity AllocateCircuit(CId, Start, End) where Start and End are Ids of end points of the circuit and CId represents the customer identifier. To design such a circuit, we need to allocate lines between end points and the nearest central offices and between the central offices (trunk connection). The latter activity may require that several hops forming a path are allocated and therefore the activity can become complex consisting of other cooperative activities. Also, the connections between the end points of the allocated lines and trunks have to be allocated in the central offices' switches to form a circuit. The example scenario is illustrated in Figure 3.

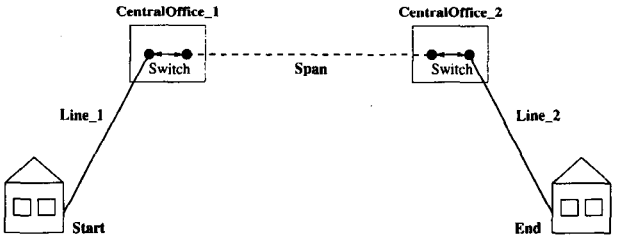

Figure 3: Example scenario

A specification language can be developed based on the VODAK Model Language [KAN94] or the language in [RB94] or [KS94]. In the example demonstrated in Table 1, we use an intuitive pseudo notation to illustrate parts of the specification of cooperative activity types. An execution scenario of this example is presented in section 3.4.

3 Support for cooperative activities

3.1 Exchanging Results of Concurrent Work

As stated earlier, our emphasis is to allow users to share and exchange the results of their concurrent work, and to guarantee that the resulting activity will be correct, i.e., obeys all constraints specified in the cooperative activity type. Since each user participating in a cooperative activity has his own private workspace, cooperation is achieved by the controlled exchange and synchronization of the contents of workspaces.

The exchange mechanisms described in the following allow the cooperating users to communicate either directly or by saving and retrieving the results of their work in/from the common activity database.

1. *Import(source_activity)*: Importing (parts of) a user activity into the workspace of the current user activity. The common activity database as well as other user activities can be addressed as *source_activity*.

2. *Delegate(dest_activity)*: Delegating the user activity from the current workspace to the workspace of *dest_activity*.

3. *Save*: Saving the current user activity in the common activity database.

CAT ALLOCATECIRCUIT(*In:* CId: CLIENT_ID, Start: POINT, End: POINT, *Out:* Circuit_ID: CIRCUIT)

Compensating activity C:
DEALLOCATECIRCUIT(Circuit_ID)

Constituent Subactivities $\mathcal{A}$:
A_0::UPDATECLIENTINFO(*In:* CId) *by* SERVICEREPRESENTATIVE
A_1::SELECTCENTRALOFFICES(*In:* Start,End, *Out:* CO_1,CO_2) *by* SERVICEREPRESENTATIVE
A_2::ALLOCATELINE(*In:* Start,CO_1, *Out:* $Line_1$) *by* ENGINEER
A_3::ALLOCATELINE(*In:* End,CO_2, *Out:* $Line_2$) *by* ENGINEER
A_4::ALLOCATESPAN(*In:* CO_1,CO_2, *Out:* Span) *by* NETWORKENGINEER
A_5::ALLOCATESWITCH(*In:* $Line_1$,Span) *by* ENGINEER
A_6::ALLOCATESWITCH(*In:* $Line_2$,Span) *by* ENGINEER
A_7::PREPAREBILL (*In:* CId,$Line_1$,$Line_2$,Span) *by* SERVICEREPRESENTATIVE

Observable States $\mathcal{O}$:
NotExecuted: $\forall A_i$NotExecuted(A_i)
Executing: $\exists A_i$Executing(A_i)
DoneWithSemanticSuccess: $\bigwedge_{i=1}^{7}$ Done(A_i))
DoneWithSemanticFailure: Done(A_0)
Undone

State transitions $\mathcal{ST}$:
NotExecuted $\longrightarrow$ Executing
Executing $\longrightarrow$ DoneWithSemanticFailure
Executing $\longrightarrow$ DoneWithSemanticSuccess

Termination states $\mathcal{T}$:
DoneWithSemanticSuccess
DoneWithSemanticFailure

Execution rules $\mathcal{E}$:
A_1 *enables* $\{A_2, A_3, A_4\}$
$\{A_2, A_4\}$ *enables* A_5
$\{A_3, A_4\}$ *enables* A_6
$\{A_2, A_3, A_4, A_5, A_6\}$ *enables* A_7
$A_0 < A_7$

Activity interleaving rules $\mathcal{I}$:

$\oplus$	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
A_0	−	+	+	+	+	+	+	−
A_1		−	−	−	−	−	−	−
A_2			−	+	+	−	+	−
A_3				−	+	+	−	−
A_4					−	−	−	−
A_5						−	+	−
A_6							−	−
A_7								−

...

EAT UPDATECLIENTINFO(*In:* CId: CLIENT_ID)
...
EAT ALLOCATELINE(*In:* P: POINT, Co: CO, *Out:* Line: LINE)
...
CAT ALLOCATESPAN(*In:* Co_1: CO, Co_2: CO, *Out:* Span: SPAN)
...

Table 1: Example of cooperative activity type specification

In the case of *Import*, a user activity is merged in accordance with the execution rules and the activity interleaving rules with the user activity of the importer. Parts of the workspace of the importing user or parts of the imported activity may need to be discarded to create a consistent "version". This operation has no effect on the workspace of the user from whom the work is being imported.

In the *Delegate* operation, the user activity of the user delegating his work is merged with the user activity of the person to whom the work is delegated. Delegation means that the user is passing work that is possibly incomplete (i.e., the work has been interrupted at one of the predefined breakpoints) to another user. This operation has no effect on the workspace of the delegatee.

The *Save* operation incorporates a user activity into the common activity database. As in the previous cases, incompatibilities between parts of the user activity and the common activity database need to be resolved under the control of the user, i.e., the responsible user has to choose between incompatible subactivities during the merge process.

All operations mentioned control the merging by the same set of rules which will be discussed in detail in section 3.2. The incorporation of activities is similar to the join operation between transactions described in [KP92] with the main difference that both the joining activity and the resulting new activity belong to the same cooperative activity. Essentially, the merging can be viewed as incorporating a partial ordered history of an activity into a partial ordered

history of a receiving activity. A partial ordered history is generated from the execution order by considering dependencies that are formed by the execution rules or by the activity interleaving rules, e.g., due to data-flow between subactivities. The merging of the histories is controlled by the partial order of each history. If the intersection of the histories that are merged is non-empty or subactivities are incompatible, some subactivities may need to be eliminated, by allowing the controlling user to choose one of the replicated subactivities.

It should also be noted, that, in principle, the *Import* and *Delegate* operations between user activities may be eliminated requiring all users to exchange information only via the common activity database. We decided to provide these operations, since their existence allows subsets of users to cooperate, which would be impossible, if these operations were not available.

Besides the mentioned operations for the exchange of information, the model provides some basic primitives to manage the execution of cooperative activities. These primitives include the instantiation of a cooperative activity, the ability to suspend a user activity at a breakpoint which can be resumed later on, and operations to join (*Join*) or leave (*Exit*) an existing cooperative activity. The *Join* operation creates a new user activity for an already existing cooperative activity while *Exit* discards the workspace of the corresponding user activity which is no longer part of the cooperative activity. Before such an operation is issued, users usually will delegate their results to another user or save them in the common activity database.

In addition, the model includes the operations *Commit*, *Abort* and *Compensate* of cooperative activities with almost usual semantics. The execution of a cooperative activity can terminate when it has reached a termination state in the common activity database. The *Commit* operation is issued by the last participating user and passes all results of the activity to the invoking activity or installs them in a public database.

3.2 Merging of User Activities

In the following we give an informal description of the basic concepts that are needed to describe a correct merger.

Activity history. With *activity history* we refer to the histories of user activities as well as to the cooperative activity history reflected in the common activity database.

An activity history is a tuple $(\mathcal{H}, \prec_{\mathcal{H}})$ where $\mathcal{H}$ is the set of all subactivities executed within the activity and $\prec_{\mathcal{H}}$ is a partial order defined over $\mathcal{H}$. A history $(\mathcal{H}, \prec_{\mathcal{H}})$ is *correct*, if it satisfies the following conditions:

1. The subactivities in $\mathcal{H}$ must be allowed by the cooperative activity type and must not violate the constraints on the number of occurrences of each subactivity in the history.
2. $\prec_{\mathcal{H}}$ must satisfy the following properties:
   (a) $\prec_{\mathcal{H}}$ must not violate the order of subactivities determined by the execution rules of the cooperative activity type.
   (b) If two subactivities $A, B \in \mathcal{H}$ are executed in the order A before B, and A is non-compatible with B, i.e., $\oplus(A, B) = \mathit{False}$, then $A \prec_{\mathcal{H}} B$ holds.
   (c) If there exist subactivities $A, B, C \in \mathcal{H}$ with $A \prec_{\mathcal{H}} B \wedge B \prec_{\mathcal{H}} C$ then also $A \prec_{\mathcal{H}} C$ holds.

The partial order $A \prec_{\mathcal{H}} B$ can be interpreted as B *depends on* A, either because there exists an execution rule in the corresponding cooperative activity type which determines the order of A and B (2a), or because A and B are not compatible, which means that the order of A and B is relevant (2b). We explicitly require the transitivity of $\prec_{\mathcal{H}}$ because transitivity is neither guaranteed by execution rules nor by the compatibility predicate $\oplus$ (2c).

A history $(\mathcal{H}, \prec_{\mathcal{H}})$ is said to be *complete* if it satisfies the termination predicate defined in its corresponding cooperative activity type. In a correct specification of a cooperative activity type a termination state can only be reached if all execution rules are satisfied (e.g., existence dependencies).

Mergeable histories. We call two histories *mergeable* if they belong to the same cooperative activity and both histories have reached a breakpoint.

Belonging to the same cooperative activity implies that the histories are executions of the same cooperative activity type. The above condition also ensures that such histories have seen the *same initial database state* because they belong to the same cooperative activity. This enables us to reduce the scope of history merging to the scope of one particular cooperative activity.

Merged activity history. A *merged activity history* is a tuple $(\mathcal{M}, \prec_{\mathcal{M}})$ that is constructed out of two mergeable activity histories $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$ and $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$ where $\mathcal{M} \subseteq \mathcal{H}_1 \cup \mathcal{H}_2$, and $\prec_{\mathcal{M}}$ is a binary relation defined over $\mathcal{M}$.

In the following, we use $(\mathcal{H}, \prec_{\mathcal{H}})$ to refer to one of the two histories $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$ or $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$ to avoid symmetric conditions. If we use $\bar{\mathcal{H}}$ and $\mathcal{H}$, we refer to different histories. A and B denote arbitrary subactivities which are included in some histories. Furthermore, we assume $A \neq B$.

The following conditions must hold for the merged history $(\mathcal{M}, \prec_{\mathcal{M}})$:

1. $\mathcal{M}$ does not violate the constraints on the number of occurrences for each subactivity defined in the cooperative activity type.

2. If $A, B \in \mathcal{H} \wedge B \in \mathcal{M} \wedge A \prec_{\mathcal{H}} B$ then $A \in \mathcal{M}$. We postulate that a subactivity B which *depends on* another subactivity A can only be included in the merged history, if A is included in the merged history.

3. If $(A \in \mathcal{H} \wedge A \notin \bar{\mathcal{H}} \wedge B \in \bar{\mathcal{H}} \wedge B \notin \mathcal{H}) \wedge \oplus(A, B) = \mathit{False}$ then $A \notin \mathcal{M} \vee B \notin \mathcal{M}$.
If there are two non-compatible subactivities $A \in \mathcal{H}$ and $B \in \bar{\mathcal{H}}$ then *at most one* of these subactivities can be contained in the merged history $(\mathcal{M}, \prec_{\mathcal{M}})$. The reason is that A and B are not independent, e.g., if A would be ordered before B in the merged history, the initial state of B in $(\bar{\mathcal{H}}, \prec_{\bar{\mathcal{H}}})$ and the initial state of B in $(\mathcal{M}, \prec_{\mathcal{M}})$ would not be semantically equivalent. If B would be ordered before A in the merged history, the execution of A would be based on a non-equivalent state.

4. $\prec_{\mathcal{M}}$ has to fulfill the following properties:

   (a) If $A, B \in \mathcal{H} \wedge A \prec_{\mathcal{H}} B \wedge A, B \in \mathcal{M}$ then $A \prec_{\mathcal{M}} B$.
   If there are two subactivities A and B from the same history $\mathcal{H}$ and $A \prec_{\mathcal{H}} B$ holds then this dependency has to be reflected in $\prec_{\mathcal{M}}$ if A and B are imported in $\mathcal{M}$.

   (b) If there exist subactivities $A, B, C \in \mathcal{M}$ with $A \prec_{\mathcal{M}} B \wedge B \prec_{\mathcal{M}} C$ then also $A \prec_{\mathcal{M}} C$ holds (transitivity condition).

Example. In the following we illustrate the merge of two histories. As stated before, two histories can only be subject of a merge, if they belong to the same cooperative activity. Therefore, they are based on the same initial database state, e.g., the state passed by an invoking activity or a public database state. Figure 4(a) reflects the initial database state of two histories $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$ and $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$. For the sake of clarity, we consider a situation where the merge of the histories is only determined by the activity interleaving rules defined in the corresponding cooperative activity type. The subactivities A and B of $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$ are to be imported into $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$. Due to the fact that $\oplus(C, B) = \mathit{False}$, rule (3) of the merged activity history states that we can include either B *or* C in the merged history. Activity A is compatible with B *and* C and thus can be easily incorporated.

Figure 4(b) continues the example. After the merge shown in Figure 4(a), subactivity D is executed in $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$, and subactivity E is executed in $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$. We now import the subactivities of history $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$

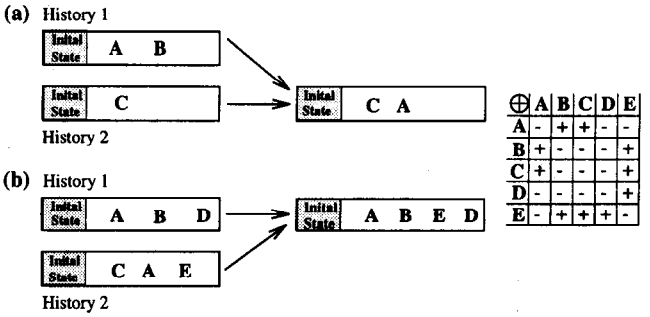

Figure 4: Merge example

into $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$. In contrast to the previous example, we choose A and B to be included in the merged history and we discard C. Now consider subactivity E. E is compatible with B, but not with A. But A is also present in history $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$. Therefore, E already depends on A, and can be incorporated into the merged history (rule (3) for merged activity histories does not apply). As E and C are compatible, E does not depend on C. Otherwise, E could not be incorporated because C is not present in the merged history. Similar arguments hold for D, which is compatible with E, but depends on A and B. As A and B are in the merge, D can be incorporated. The correctness of the activity histories $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$ and $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$ together with the merger rules ensure the correctness of the merged history shown in Figure 4.

In the following, we outline a few important properties of the proposed model:

1. Histories resulting from a merge contain only subactivities that are allowed by the corresponding cooperative activity type.

   Condition (2) of merged activity histories guarantees that a merged history $(\mathcal{M}, \prec_{\mathcal{M}})$ contains only activities that are either in $\mathcal{H}_1$ or $\mathcal{H}_2$. Due to the fact that $\mathcal{H}_1$ and $\mathcal{H}_2$ satisfy condition (1) of correct activity histories they can only contain subactivities that are in line with the specification.

2. A merged history $(\mathcal{M}, \prec_{\mathcal{M}})$ does not violate the order of subactivities determined by the execution rules of the cooperative activity type.

   In both histories $(\mathcal{H}_1, \prec_{\mathcal{H}_1})$ and $(\mathcal{H}_2, \prec_{\mathcal{H}_2})$ the execution dependencies are not violated by $\prec_{\mathcal{H}_1}$ or $\prec_{\mathcal{H}_2}$, respectively.

   If A and B are from the same history $(A, B \in \mathcal{H})$ and $A, B \in \mathcal{M}$ and the execution dependencies determine an order A before B, then $A \prec_{\mathcal{H}} B$ and therefore $A \prec_{\mathcal{M}} B$ is satisfied by condition (4a) of merged activity histories.

Assume A and B come from different user histories, i.e., $A \in \mathcal{H}$, $B \in \bar{\mathcal{H}}$. If there exists an execution rule in the cooperative activity type which orders A before B, then it holds by the properties of correct activity histories that A is also included in $\bar{\mathcal{H}}$, i.e., $A \in \bar{\mathcal{H}}$ and $A \prec_{\bar{\mathcal{H}}} B$. Thus, by condition (4a) of merged activity histories, $A \prec_{\mathcal{M}} B$ follows.

3. Each subactivity $A \in \mathcal{M}$ in a merged history $(\mathcal{M}, \prec_{\mathcal{M}})$ is executed based on a state that is equivalent to the state before its execution in the original history.

   The relation $\prec_{\mathcal{M}}$ preserves the local partial orders $\prec_{\mathcal{H}_1}, \prec_{\mathcal{H}_2}$ of histories. As stated above, the execution rules defined in the cooperative activity type are satisfied. According to rule (3) for merged activity histories, for non-compatible pairs of subactivities of different histories that are not contained in both histories, one of the subactivities is discarded and is not included in the merged history. This ensures that the equivalence of the initial states of a subactivity in its original history and in the merged history is not affected by incorporating subactivities of the other history.

The required properties for merged activity histories can be interpreted as a *test on the correctness* of a history that contains subactivities from different histories, i.e., the merged history. We have outlined above that merged histories fulfill the properties required for correct activity histories. Due to the fact that a merged history fulfils the properties of correct activity histories, the merged history could have been executed by a single fictitious user.

In practice, we need a constructive algorithm for different merge processes, e.g., import, delegation, or save of work in the common activity database. These algorithms will be based on the outlined properties of merged activity histories.

The selected approach, i.e., to define the correctness of a merged history by assuming two correct histories as input, has the advantage that the scheduling of subactivities within user activities (to ensure the correctness of single activity histories) is independent of the merge algorithm itself. Therefore, it is possible to use different software modules for the enforcement of execution rules and for merging of histories, i.e., exchanging information between different user activities or a user activity and the common activity database.

3.3 Coordination Facilities

Coordination facilities serve the purpose of coordinating cooperating users by providing a set of functions implementing predefined coordination protocols. Table 2 summarizes some functions useful to coordinate the processing of cooperative activities.

Preempt
Request exclusive access
Request cooperation
Request observation

Table 2: CoAct coordination functions

The functions are available at the user interface level and are independent of a particular cooperative activity type. They allow e.g., to join an ongoing cooperative activity or to observe concurrent user activities. If one of these functions is invoked, the system can (depending on the user types) accept or refuse the execution. For some cases approval of users working in the same cooperative activity is necessary.

If for some reason the participation of a specific user is no longer desired, he can be eliminated from the cooperative activity and his work can be taken over by another user. This is done by executing the *Preempt* function. A user can only be forced to leave the cooperative activity, if the user types do not warrant such an interaction. In this case the request is rejected by the system.

The *Request exclusive access* function is executed by a user who wants to take over a co-worker's activity by merging it with his own user activity. If this function is invoked by a user who is not participating in the cooperative activity, the workspace of the current user is handed over to the requestor. The function gives the affected user the ability to decide whether he grants the exclusive access or not. Before the affected user grants the exclusive access he can request who wants to take over his activity.

To join an ongoing cooperative activity the *Request cooperation* function is issued. This request can be refused by the actual participants. Furthermore, this function allows to invite a new person to contribute to the cooperative activity.

Different notification policies can be selected with the *Request observation* function. Once different user activities are active, the notification policy determines which particular events are subject to notification. In particular, collaborating users are informed when the state of a workspace of another user changes, e.g., a breakpoint is reached within a concurrent user activity. The collaborating users are also informed when the state of the common activity database changes, i.e., one of the users performs the *save* operation. Hence, this mutual awareness helps the users to coordinate their current activities and helps to avoid loss of work due to incompatibilities in a later merge process. For monitoring reasons, it is useful to provide this operation to persons, e.g., a division manager, who only observe the progress of the cooperative activity.

3.4 Example of the Execution of a Cooperative Activity

Let us consider the following scenario in which four different users participate in the execution of a cooperative activity of type ALLOCATECIRCUIT described in table 1. The scenario illustrates the basic concepts introduced in this paper, namely breakpoints, execution rules, interaction rules, the merge option, and coordination facilities. We assume that a cooperative activity is uniquely identified. In our example all users cooperate by performing various subactivities related to the execution of a single service order.

1. $User_1$ starts a cooperative activity of type ALLOCATECIRCUIT upon receiving a service order and he gets a unique identifier for this cooperative activity, e.g., C_1. He executes, as part of his user activity, subactivity a_0 of type UPDATECLIENTINFO and a_1 of type SELECTCENTRALOFFICES; after completion of these subactivities, his user history looks like that:

   $User_1$: `[ a0, a1 ]`

   The user then executes a *save* operation: the state of his user activity corresponding to a breakpoint is saved in the common activity database. No explicit merge has to be performed because the common activity database is empty. Afterwards, the user exits the cooperative activity.

2. $User_2$ comes and joins the cooperative activity C_1. He is informed that C_1 was being executed by $User_1$ and that $User_1$ has saved his work in the common activity database. $User_2$ decides to *import* the current state of work from the common activity database. He then executes a_2, ALLOCATELINE and a_4, ALLOCATESPAN.

   $User_2$: `[ a0, a1, a2, a4 ]`

3. $User_3$ comes and joins cooperative activity C_1. He is informed that $User_2$ is active in the cooperative activity C_1, requests to import (which is approved) and imports $User_2$'s history. Afterwards, he executes a_3, ALLOCATELINE followed by a_5, ALLOCATESWITCH for $Line_1$ and a_6, ALLOCATESWITCH for $Line_2$

   $User_3$: `[ a0, a1, a2, a4, a3, a5, a6 ]`

4. In the meantime, $User_2$ executes a_5', ALLOCATESWITCH for $Line_1$ and decides to delegate his work to $User_3$. The history of $User_2$ is shown below. After the delegation is accepted by $User_3$, $User_2$ performs an exit.

   $User_2$: `[ a0, a1, a2, a4, a5' ]`

5. $User_3$ accepts the work being delegated and it is merged into his user history. The histories are mergeable. But there is a problem with the two connections between $Line_1$ and *Span*, which were made independently (a_5, a_5'). $User_3$ chooses to accept the subactivity a_5' originally executed by $User_2$. This means that "his version" of a_5 is discarded, i.e., compensated, and the subactivity a_5' is redone in $User_2$'s private workspace.

   $User_3$: `[ a0, a1, a2, a4, a3, a6, a5' ]`

6. $User_4$ (Manager) requests exclusive access to C_1 and imports $User_3$'s user activity workspace. $User_3$ leaves the cooperation and his workspace is discarded. $User_4$ decides that a_2 is not to his liking. He re-executes a_2', as a result the old a_2 and a_5' are invalidated because a_2 and a_2' are non-compatible and a_5' depends on a_2. $User_4$ re-executes a_5'', prepares the bill (a_7) which completes the execution of the cooperative activity C_1 and saves his user activity to the common activity database. Since a termination state of C_1 has been reached (DoneWithSemanticSuccess), the user can issue a *commit*, making the results of the cooperative activity visible outside. The final cooperative activity history ($User_4$'s history respectively) is shown below.

   Final C_1 history:
   `[ a0, a1, a4, a3, a6, a2', a5'', a7 ]`

We see that no execution rules have been violated and the final result of the cooperative activity C_1 is equivalent to the execution by a single fictitious user in the order $[a_0, a_1, a_4, a_3, a_6, a_2', a_5'', a_7]$ without any improper interferences form concurrent execution of user activities and with a proper data-flow.

4 Related Work

There have been various approaches proposed to increase the flexibility of the transaction concept to support cooperation.

The basic check-out model is a commonly accepted approach to support long interactive computations [BKK85, KKB87, KSUW85]. Users copy objects from a shared database to private workspaces for manipulation (check-out). Check-out objects are reserved for exclusive access until a later check-in. Another drawback is that all objects and subobjects needed to perform a specific task have to be checked out explicitly. The approach is often combined with versioning mechanisms. Concurrently derived versions have to be merged manually.

For software engineering purposes, several extensions of the check-out model have been developed taking advantage of the opportunity of generic consistency checking in those environments [KF87, KPS89, Hon88]. The Network Software Environment [Hon88] supports an optimistic coordination scheme by forcing concurrently created versions of source code files to be merged

(copy/modify/merge cycle). The merge operation includes an automatic consistency checking for the affected files. Due to the fact that such a generic consistency checking is not applicable in all application domains, this approach is not a general solution.

The split- and join-transaction approach [PKH88] supports the restructuring of transactions. A split-operation allows to split a running transaction into two new (serial or independent) transactions while 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.

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 lock schemes [KSUW85] or by following predefined access patterns that define the application-specific correctness criteria [FZ89]. The models pass objects along the transaction group hierarchy.

ACTA [CR90, CR92] is a comprehensive transaction framework that can be used to specify the types of dependencies between transactions. One of the ACTA building blocks is the delegation primitive [CR93]. A transaction t can delegate the responsibility for committing or aborting an operation o to another transaction t'. This primitive is useful to control the (partial) visibility of results.

The area of transactional workflow approaches usually comprises task specification languages to express various execution constraints for a set of 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 [DHL90]. Cooperation is characterized in these models by passing results between workflow tasks in a predefined manner. There is no opportunity for passing results back and forth between co-workers to achieve a common goal. This is needed in non-workflow scenarios, e.g., cooperative authoring environments.

Related work done in the field of Computer Supported Cooperative Work [WL93, GS87] follow in most cases a more or less ad-hoc approach in synchronizing work of collaborating users, e.g., user-controlled locking of objects without transactional support. The requirement of supporting different modes of cooperation [HW92] can be supported in the CoAct model by defining different sets of execution rules and activity interleaving rules.

5 Conclusion and Further Work

In this paper, we proposed the CoAct model as a possible approach towards transactional support for cooperative applications.

Cooperation in our model is achieved by controlled exchange and synchronization of the contents of workspaces among the users and by installing results of their activities in the common activity database. While the traditional transaction model guarantees that no errors occur due to the interleaved execution of transactions (serializability), the outlined correctness criterion of our model is based on two assumptions:

1. Histories of user activities as well as the history reflected in the common activity database are correct in the sense that they satisfy all execution rules defined in the cooperative activity type, and

2. there will be no inconsistencies introduced due to the exchange of information between concurrently executed activities. The exchange of information by means of merging activities is based on the semantic compatibility of activities described in the activity interleaving rules.

Furthermore, we allow only a single result of a cooperative activity although there may exist various versions in the different user workspaces and the common activity database, respectively. Thus, the CoAct model ensures that the joint execution of a cooperative activity is equivalent to one that could have been carried out by a single user.

The major contribution of the presented model is the combination of modelling primitives as found in the workflow area and primitives as needed in less structured cooperative environments like cooperative authoring. By utilizing application semantics, our approach shows how the work of different cooperating users following a common goal can be merged in accordance with the given rules of a specific application domain. With the merge option, co-workers can delegate parts of their work as the need arises as well as create their own realities of the given problem.

So far, we have worked out a framework of a cooperative activity model. Our further research focuses on carrying out detailed algorithms and a theoretical framework to prove the properties of the model. In addition, we work on embedding the CoAct modelling facilities in the object-oriented database programming language VML and extending the transactional services of the OODBMS VODAK [GI95] with the CoAct functionality to support cooperation.

References

[ANRS92] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using flexible transactions to support multi-system telecommunication applications. In *Proc. of the 18th Int. Conference on Very Large Databases*, 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 *Proc. of the 19th Int. Conference on Very Large Databases*, pages 134–145, Dublin, Ireland, August 1993.

[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.

[CR90] P. K. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 194–203, Atlantic City, NJ, USA, May 1990.

[CR91] P. K. Chrysanthis and K. Ramamritham. A formalism for extended transaction model. In *Proc. of the 17th Int. Conference on Very Large Databases*, pages 103–112, Barcelona, Catalonia, 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.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 204–214, Atlantic City, NJ, USA, May 1990.

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

[FZ89] M. F. Fernandez and S. B. Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Proc. of the Int. Workshop on Persistent Object Systems*, pages 341–350, January 1989. Newcastle, New South Wales.

[GHKM94] D. Georgakopolous, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proc. of the 10th IEEE Int. Conference on Data Engineering*, pages 462–473, Houston, Texas, February 1994.

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

[GS87] I. Greif and S. Sarin. Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2):187–211, April 1987.

[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.

[HR83] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[HW92] J.M. Haake and B. Wilson. Supporting collaborative writing of hyperdocuments in SEPIA. In *Proc. of the ACM Conference on Computer-Supported Cooperative Work*, pages 138–146, Toronto, Canada, October 1992.

[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. Öszu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, NATO ASI Series, pages 389–443. Springer Verlag, June 1994.

[KF87] G. E. Kaiser and P. H. Feiler. Intelligent assistance without artificial intelligence. In *Proc. of the 32th IEEE Computer Society International Conference*, pages 236–241, San Francisco, California, USA, February 1987.

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

[KKB87] H. F. Korth, W. Kim, and F. Bancilhon. On long-duration CAD transactions. *Information Systems*, 13, 1987.

[Kle91] J. Klein. Advanced rule driven transaction management. In *Proc. of the 36th IEEE Computer Society International Conference*, pages 562–567, San Francisco, California, USA, March 1991.

[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.

[KS94] N. Krishnakumar and A. Sheth. Specifying multi-system workflow applications in METEOR. Bellcore, Technical Memorandum, 1994.

[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.

[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.

[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.

[RS95] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In Kim [Kim95], chapter 29, pages 592–620.

[SANR92] A. Sheth, M. Ansari, L. Ness, and M. Rusinkiewicz. Using flexible transactions to support multidatabase applications. In *US West - NSF - DARPA Workshop on Heterogeneous Databases and Semantic Interoperability*. Boulder, Colorado, USA, February 1992.

[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, 1993. Seattle, Washington, Nov 14–18.

[WR92] H. Wächter and A. Reuter. The ConTract model. In Elmagarmid [Elm92], chapter 7, pages 219–264.