Transaction-oriented Work-flow Concepts in Inter-organizational Environments*

Jian Tang† and Jari Veijalainen
VTT Information Technology
Multimedia Systems P.B. 1203
FIN-02044 VTT, Finland
email: jian@cs.mun.ca, jari.veijalainen@vtt.fi

Abstract

Workflow techniques have gained a lot of attention as a means to support business process re-engineering but also as a means to integrate legacy systems. Most workflow models view the applications as a fixed set of tasks. In this paper we analyse inter-organisational application domains and analyse properties for transactional workflows and systems supporting them. We study a workflow model for the applications where job step instances cannot be fixed in advance. We analyse requirements arising from this kind of environments through a particular application, and introduce special modelling components to support these requirements. We also develop the concept of C-unit to cope with concurrency anomalies and recovery.

1 Introduction

Modern business application systems are usually composed of independently developed smaller components, and be accessed by concurrent users. Due to their enormously diversified nature, these systems may have very different requirements and exhibit a quite complicated internal structures. Under such an environment, therefore, the productivity and the effectiveness of business processing is of a particular concern.

Workflow techniques were first developed for and used in distributed project management systems. The background were big government funded software projects in USA in the seventies [15]. Currently, they are seen as the key techniques to facilitate business process re-engineering by supporting process-oriented work-groups.

A *workflow specification* consists of type descriptions of the jobs steps, the relationship, commonly called *dependencies*, among job steps, and, additionally, their execution requirements. A job step defines some work to be done. From the point of view of the global application semantics a job step is indivisible. Within the local systems, however, it may have a more complicated structure. A *workflow* is a finite collection of job steps organised to accomplish some business process.

The workflow tools are used to specify workflows. They are part of a new set of products and resources for solving business problems. Most of the current tools, however, still lack many features like transactional properties at the workflow level and the conceptual background of the products is rather ad-hoc in nature. One exception, however, is the IBM FlowMark product [16], which is based on coloured Petri-nets as the description formalism. In the development of the business products also new results on the transaction models will soon be taken into account [6, 5].

The early workflow models did not address data sharing, persistence and failure recovery. Recently, a number of *extended transaction models* [7, 9, 11, 17, 18] have been defined (for an overview, see [8, 10]). These models focus on selective relaxation of atomicity or isolation in order to better match the requirement of some database applications which the traditional ACID transactions are too restrictive to satisfy. As a result, the performance of the workflow system is improved and some special features, such as cooperation and long running activities, can be supported. In addition, to a limited degree, these models address the problems arising from heterogeneity and/or autonomy in component systems. These models belong to a special class of workflow models, called *transactional workflows*.

The models developed in [4, 13, 12] have a different flavour from the above in that they provide a common framework within which one can specify and reason about the nature of interactions between transactions in a particular model. For this reason, they are sometimes called *transaction meta-models*. In [14], the authors give a survey of the infrastructure of the current workflow systems. In [1, 2, 3, 20], the authors discuss issues relating to workflow systems implementation.

Most of the existing models view a workflow application as a fixed set of tasks. The advantage of this view is that it can simplify the specification of the requirements. As a direct benefit, all the possible dependencies are known in advance and can be directly specified. The limitation of such a view is that it does not satisfy needs of more dynamic applications, such as the one discussed in this paper. In addition, the issue of consistency in concurrent workflow execution has not drawn adequate attention.

In this paper, we study a workflow model for the applications where job steps cannot be fixed in advance. We analyse requirements arising from this kind of environment

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

†The work was performed while the author was visiting VTT, Espoo, Finland, and subsequently Industrial Technology Research Institute, Hsinchu, Taiwan. On leave from Dept. of Computer Science, Memorial University of New Foundland, Canada.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
CIKM '95, Baltimore MD USA
© 1995 ACM 0-89791-812-6/95/11..$3.50

through a particular application, and introduce special modelling components to support these requirements. We also study an approach to handling interleaving problem in the face of concurrent execution of multiple workflows.

The rest of this paper is organised as follows. In Section 2, we introduce a practical application environment which motivates our work. In Section 3, we discuss the overall model supporting various requirements arising from the introduced application and the like, its special features and also some well-known features from other inter-organisational environments. In Section 4, we discuss the C-units and in Section 5 we outline the system architecture. We conclude the paper by summarising the main results.

2 The PortNet system

In this section, we describe (partially) a real system which will be used as an example throughout the remainder of this paper to explain the main concepts in our model, PortNet [21]. It has been in the making at Finnish ports, at some governmental instances (such as National Board of Navigation and National Board of Customs), and at some stevedoring companies in ports, since 1992. It is to handle notices of vessel arrival and departure. The goals of PortNet are to simplify the current business procedures of the participants, to create a uniform business process model for all Finnish ports and to gain reduced cost as well as other benefits. The current system is EDI-based, i.e. the interfaces between different participants are specified using the standard EDI language.

Relating to the visit of a foreign ship are various kinds of activities to be carried out by a number of departments or other organisations. For example, the activities relating to the ship arrival may include declaration to the customs, recording information concerning the arrival date, the cargo, and the ship itself into various databases, navigation of the ship and other services, etc. The activities relating to the ship departure may include recording information concerning the departure date into the databases, navigation of the ship and other services, calculating the duration of the stay of the ship and the related cost, etc. In the following, we will describe these activities in more detail. Due to the space limitation and the similarities between the activities for ship arrival and ship departure, we will only concentrate on the activities for ship arrival, but omit those for ship departure.

The processing of a ship visit starts when an agent receives a manifest from a foreign agent. When this happens, or when the agent receives the further notification (See explanations later in this section.) from the foreign agent, s/he will create an EDI message called Advanced Arrival Notice (AAN) and initiates the following process:

Preparing Adv. Arrival Notice by Agent (PRE-AAN-AGENT)

1. If this process is initiated as a result of receiving a further notification, then go to 5;
2. Forward the manifest to a stevedoring company;
3. Send the local customs office the "vessel declaration for the port" form, a notice of ship arrival and a "vessel declaration for the customs" form;
4. Record the information about the service order into the database;
5. Record the information about the vessel (name, radio call sign, length, width, machine power, etc.), the estimated arrival date, and the dangerous goods the ship may carry, into the local database;
6. Send an Advance Arrival Notice including all the information described at step 5 to the port and the National Board of Navigation (NBN);

END PRE-AAN-AGENT

When the port receives the AAN from the agent or NBN, or when the appointed person at the port obtains an AAN from other (supposedly more reliable) source, the following process is initiated:

Processing Advance Arrival Notice at port, (PRO-AAN-PORT)

1. Update the arrival list of vessels: if the input is concerned with a vessel that has not been in the arrival list, then insert it;
2. Enter all the information contained in the notice, including those about vessels, the estimated arrival date and the goods they carry into the database. If this is not the first time these kinds of information are entered for any particular ship, then the new information must overwrite the old information and mark the status code for those information as "changed";
3. Call for local service companies to prepare the service for the arriving ship;
4. If this process is initiated as a result of the appointed person receiving an AAN from other sources then send this AAN to the NBN;

END PRO-AAN-PORT

When the NBN receives the AAN from the agent or the port, or when the appointed person at the NBN obtains an AAN from other (supposedly more reliable) source, the following process is initiated:

Processing Advance Arrival Notice at NBN, PRO-AAN-NBN

1. same as step 1 in PRO-AAN-PORT;
2. same as step 2 in PRO-AAN-PORT;
3. Send an Advance Arrival Notice to the pilot station to prepare for ordering a pilot to navigate the ship to the port;
4. If this process is initiated as a result of the appointed person receiving an ANN from other sources, then send this AAN to the port;

END PRO-AAN-NBN

When the ship actually arrives at the port, the agent creates a Final Arrival Notice (FAN) and initiates the following process:

Preparing Final Arrival Notice by Agent, PRE-FAN-AGENT

1. Enter the final arrival date of the ship into the database;
2. Send the Final Arrival Notice to the port and the NBN;

END PRO-FAN-AGENT

When the port receives the FAN from the agent, the following process is initiated:

Processing Final Arrival Notice at port (*PRO-FAN-PORT*)

1. Enter the final arrival date of the ship into the local database;

2. Finalise the service date to the local service company;

END PRO-FAN-PORT

When the NBN receives the FAN from the agent, the following process is initiated:

Processing Final Arrival Notice at NBN, PRO-FAN-NBN

1. Enter the final arrival date of the ship into the local database;

2. Finalise the navigation date to the pilot station;

END PRO-FAN-NBN

The agent may receive further notification from the foreign agent from time to time because the information contained in each notification is only an estimation in nature and may have to be refined repeatedly. For example, the exact date of arrival of a ship is affected by many unpredictable factors, such as the weather conditions on the ocean, the possible delay in the intermediate ports, etc. As the exact arriving date draws closer, the information will become more accurate.

At the port and NBN there is an appointed person who may receive AAN from more reliable sources than the agent, for example, the captain of the ship. When this happens, the information is considered to be more accurate than those obtained from the agent.

When the ship actually arrives at the port, the agent initiates "Preparing Final Arrival Notice by Agent", and it is required that this process finish no later than three days after the arrival of the ship. In the process, a Final Arrival Notice is sent to both the port and the NBN to initiate *PRO-FAN-PORT* and *PRO-FAN-NBN*, respectively. Once *PRO-FAN-PORT* (*PRO-FAN-NBN*) is initiated, any ongoing execution of *PRO-AAN-PORT* (*PRO-AAN-NBN*) is no longer necessary and, therefore, should stop.

3 The general structure of the transactional workflow model

In this section, we describe the essential components of our model. We first outline the basic concepts that are common in several existing models. We then introduce the new features contained in our model.

3.1 Job steps and dependencies

The model consists of two parts, the workflow specifications and workflow instances, also called workflows. A workflow specification consists of a finite number of *step plans* and *dependencies* between the *job steps*. A job step is a minimal unit work (computation) that defines a semantically meaningful portion of the overall work, and therefore is indivisible from the point of view of the workflow application. A workflow consists of a number of job steps, which are generated by executing the corresponding step plans. Each execution of of a step plan results in a new job step.

A novel feature in our model is that a step plan may be invoked more than once in a single workflow. Different invokers may be associated with a step plan. The way an invoker is defined is flexible, depending upon the requirements of the application.

For example, in PortNet, *PRE-AAN-AGENT*, *PRO-AAN-PORT*, $\cdots$, are all step plans, and their executions generate job steps. We can associate three invokers with *PRO-AAN-PORT*, the agent, the appointed person at NBN and the appointed person at the port. Note that the former two invokers invoke the step plan only *indirectly*, i.e., by sending messages, and the latter is a direct invoker.

Each job step has a number of states. Typical states include *start*, indicating the start of a job step; *not-start*, indicating the job step has not been started yet; *executing*, indicating the state of a job step after it has started but before it terminates; *commit*, signifying that all the operations in a job step have been executed successfully and their effects have been irrevocably stored in the system; and *abort*, signifying the failure of the job step, which requires that all the effects of the job step be eliminated as if it had never been executed.

The states of different job steps may be related by *state dependencies*. A state dependency specifies constraints on job steps' entering certain states. Let s_1 and s_2 be two job steps. The following are some of the frequently encountered state dependencies:

BB s_2 can start only after s_1 starts–s_1 BB s_2 (Begin-on-Begin);

BC s_2 can start only after s_1 commits– s_1 BC s_2;(Begin-on-Commit)

BA s_2 can start only after s_1 aborts–s_1 BA s_2 ;(Begin-on-Abort)

CC If s_1 and s_2 both commit, then s_2 commits after s_1 does–s_1 CC s_2 ;(Commit-Commit)

AB If s_1 starts, then s_2 aborts–s_1 AB s_2 ;(Abort-on-Begin)

WAB If s_1 starts, then if s_2 has not committed yet, it must abort–s_1 WAB s_2 ;(Weak-Abort-on-Begin)

Various state dependencies exist among the job steps in PortNet. Shown in Figure 1 are those related to the job steps which handle arrival notices. In the figure, larger and smaller rectangles represent step plans and job steps, respectively. A smaller rectangular inside a bigger one represents the job step generated by the corresponding step plan. Thus a rectangular with multiple smaller rectangles being inside it indicates the corresponding step plan has been invoked more than once and therefore multiple job steps have been generated. The dependencies shown are based on the requirements described in section 2. For example, since any ongoing process handling AAN at the port must abort once the processing of FAN starts at the port, the former has a dependency WAB on the latter. Note that the arrows for dependencies BB and CC are coarse grained notations, which are refined in Figure 2.

In Figure 2, the numbers inside small rectangles denote the order in which the corresponding step plans are invoked to generate the job steps. Since the later invocations of a step plan process more recent (i.e, more accurate) AANs than the preceding ones, the later job steps must overwrite the earlier ones to store AANs. This requirement could be implemented by a Commit-Commit dependency between the two in the PortNetFor the step plans PRO-AAN-PORT and PRO-AAN-NBN, the job steps i and j, respectively, are invoked by the appointed persons at the port and NBN. Since the appointed persons presumably obtain AAN from the more reliable sources, these job steps should overwrite the job steps invoked by the agent[1].

[1] A problem arises when both PRO-AAN-PORT and PRO-AAN-NBN are invoked by the respective appointed persons. Since both of

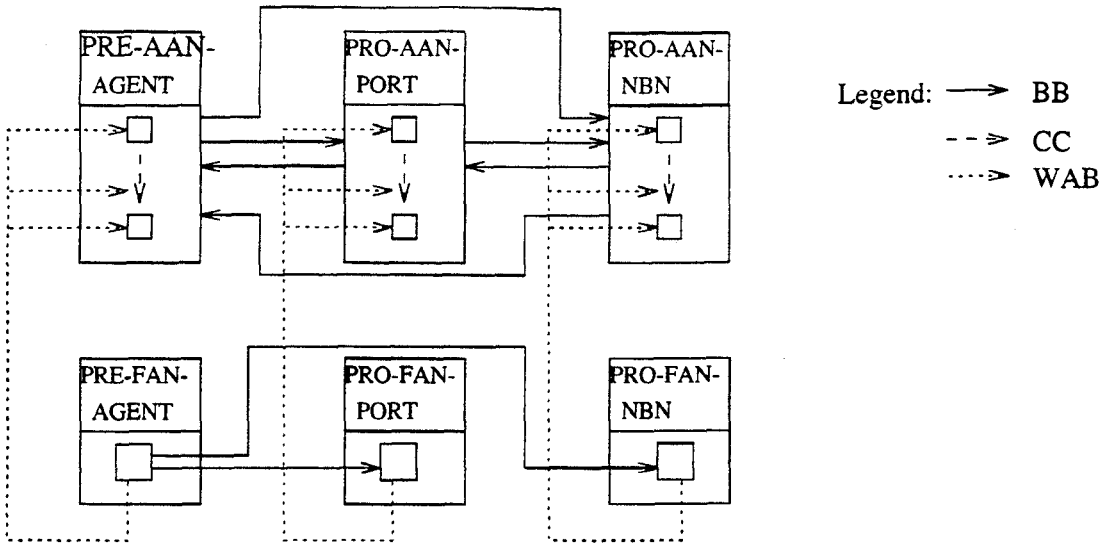


Figure 1. dependencies among job steps for handling arrival notices.

Another kind of dependency, called a *value dependency*, defines *data flow* between job steps. A job step is value dependent on another job step if the results generated by the former depends on the value generated by the latter. For example, a job step in *PRE-AAN-AGENT* will send an AAN to the port to invoke *PRO-AAN-PORT*, resulting in a job step which processes the received AAN. Therefore, the later has a value dependency on the former.

Value dependency between job steps usually also induces an *ordering* between them, because the dependent cannot start before the value has been delivered by the other. Many workflows also require the support of *temporal dependencies*. A temporal dependency imposes time constraints on a job step. One such example in PortNet is that the job step generated by *PRE-FAN-AGENT* must finish no later than three days after the arrival of the ship.

Like some extended transaction models, we allow *compensable job steps*, which can commit before the workflow terminates. If the workflow eventually fails, a committed compensable job step will be compensated for by a *compensating job step*. If a job step fails, it may be *retried*, i.e., the step plan is resubmitted with the identical inputs. Alternatively, another functionally equivalent job step, called its *alternative*, may be activated to finish the desired task.

3.2 Acceptable states of workflows

The inclusion of compensating and/or alternative job steps implies that the entire workflow may fail even if some constituent job steps commit, or it may be successful even if some of them fail. This property is characterised by the notion of *acceptable states* (cf. semantic constraint in [22]). An acceptable state of a workflow is a collection of commit and abort states of the member job steps which signifies the fulfilment of the goal of the workflow. Note that acceptable states must be defined by workflow designers.

PortNet example above can use relatively simple acceptable states, since we do not include alternative job steps. Thus each step plan must have a successful execution. (This can be achieved e.g. by resubmission.)

them are supposedly to get their information from reliable sources, which one is more accurate? An approach to coping with this problem is to prioritise all possible sources, so that a higher priority signifies a higher accuracy. Due to space limitations, we will not pursue this issue further.

3.3 Job step identification

Job step identification is necessary because, among other things, they are related by dependencies which must be specified by the application and enforced by the supporting system. If all job steps which have dependencies among them are known at the specification time, like what most of the existing models assume, then identifying job steps is simple. For example, in the well known "make a trip" example, reserving a seat in any particular airline, renting a car from any particular car-rental company, and booking a room in any particular hotel, all specify a single job step. Thus job step identification is trivial: each job step is represented by the corresponding step plan. However, there are applications where all job steps are not known in advance. For example, in PortNet, step plan "PRE-AAN-AGENT" is invoked each time the agent gets further notification. It is not known in advance how many notifications the agent will receive. In addition, dependencies exist among the job steps resulting from different invocation of this step plan, as well as between them and other job steps. This makes some kind of referencing mechanism necessary for individual job steps generated by the same step plan. (See the examples in Section 3.2.2.)

3.4 Mechanisms

Our goal is to provide an ability to reference any individual job step even if all job steps are not known in advance. In addition, we also want to be able to reference any group of job steps, because there might be dependencies specifiable for groups. For this purpose, we incorporate two data types into the model, *group* and *sequence*. A group is a named collection of job steps. The job steps in a group may be generated by the same or different step plans.

Besides the normal 'set operations', we also allow a new class of operations to be defined on a group. These operations return a subset of the group which consists of the job steps with some common properties, such as the invoker, the time of invocation, etc. Also operations must be used to explicitly assign names to groups. For simplicity, we use

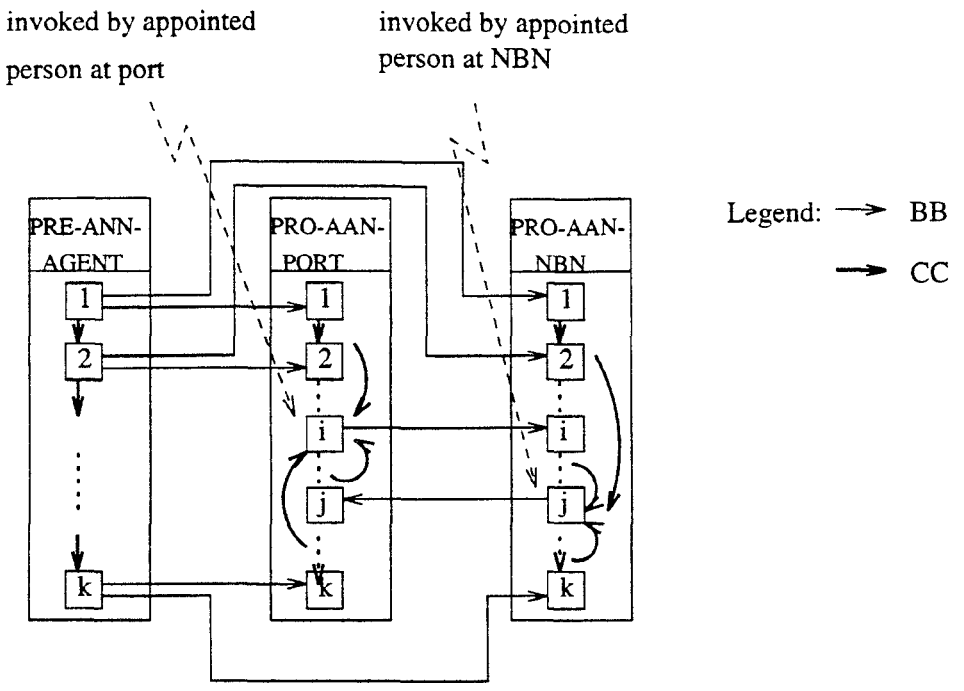


Figure 2. Dependencies among job steps for preparing/processing advanced arrival notice.

the convention that the name of a step plan is a name of the group consisting of all the job steps generated by that step plan. A requirement of the group type here is that each time a job step is generated, it is inserted automatically into the group(s) it should belong. For example, if g is defined as the group of all committed job steps generated by a step plan C, then whenever the invocation of C leads to a successful execution, g will automatically contain one more job step corresponding to that execution.

A sequence is a named collection of job steps which are ordered based on some well defined criteria observable externally. For example, we can order the job steps generated by the same step plan into a sequence according to the time the step plan is invoked. As another example, if a collection of job steps generate comparable values, then they can form a sequence based on the values they generate.

Before a sequence can be referenced, it must be *created*. A sequence is created by performing a special operation on a group. The operation must specify the rules based on which the elements are ordered. If a sequence s is created from a group g, then any job step that later joins g, through further invocations, for example, will be automatically inserted into s at the proper position.

The operations defined on a sequence in most cases are used to identify specific elements, such as the first, the last, or in general the ith element. Operations can also return the number of elements in a sequence. In addition, a sequence must be named explicitly by performing some special operations.

From the above discussion, the sequences and the groups defined by our model differ from the traditional sequences and groups in that in our model, the sizes of sequences or groups may increase "by itself" as time goes by, while a normal sequence or group has a fixed size unless some operations are performed explicitly on them (for instance, inserting elements into sequences, union of groups, etc.) which result in the change of the size.

Based on these two types, we can specify dependencies among arbitrary job steps by using a *dependency specifier*. A dependency specifier can be viewed as a predicate, in which the "variables" are the operators on the data types. A dependency specifier specifies a condition that must be met by the workflow. In the following, we use examples to explain the concepts introduced earlier. Note that it is not our intention to introduce any specific language structures. What we want is to demonstrate how these data types make it possible to specify dependencies in a broad domain.

Example 1: As mentioned before, in PortNet since the job steps resulting from earlier invocations of PRE-AAN-AGENT process less accurate information than those from later invocations, we define a dependency CC from the former to the latter, making the effects of the latter overwrite the effects from the former. Thus, we first arrange all the job steps generated from this step plan as a sequence based on the time of invocations.

$<Seq>$:= CRO $<PRE\text{-}AAN\text{-}AGENT>$;

We then use the dependency specifier to specify the dependency:

$\forall i, j$, GET($<Seq>, i$) CC GET($<Seq>, i+j$)

In the first statement, operator CRO creates a sequence $<Seq>$ from group $< PRE-AAN-AGENT >$. The order imposed on the elements of $<Seq>$ is their chronological order of invocation. In the second statement, GET is an

operator that selects the ith and $i+j$th elements of <*Seq*>. The statement means that any job step in sequence <*Seq*> has a dependency *CC* on its predecessor.

Example 2: As explained before, the job step of PRO-AAN-PORT invoked by the appointed person at the port has a *CC* on those of the same step plan invoked by the agent. This calls for the identification of the group of all job steps generated through a particular invoker.

Group1 := GRO (INVOKER(<*PRO-AAN-PORT*>, *appointed-person-at-port*));
Group2 := GRO (INVOKER(<*PRO-AAN-PORT*>, *agent*));
Group2 CC Group1;

Operator INVOKER creates a group consisting of all job steps resulting from the invocation by specified invokers. Operator GRO allows a group to be named. In the third statement, *CC* is specified between two groups, which means that any two job steps of one group each must satisfy the dependency.

Note that although the dependency specifier described by the third statement looks identical to the dependency specification in many existing models, the semantics here has been extended from a single pair of job steps to multiple pairs.

Example 3: As illustrated in Section 2, it is required that once any job step in PRE-FAN-AGENT starts, no job step in PRE-AAN-AGENT can commit.

Group1 := GRO (<*PRE-AAN-AGENT*>);
Group2 := GRO (<*PRE-FAN-AGENT*>);
Group2 WAB Group1;

Recall that *WAB* is Weak-Abort-on-Begin dependency. The dependency specifier of the last statement is interpreted as: for any job step in *Group2* and any job step in *Group1*, if the former starts, then if the latter has not committed yet, it must abort.

A simple dependency specifier can be used conjunctively with other dependency specifiers or traditional predicates to form a composite dependency specifier. Composite dependency specifiers can specify more complicated dependencies, as the following example shows.

Example 4: In many workflow application, compensating transactions are repeatedly submitted until they commit. The following is a specification in such a situation. Let *C* be a program that defines a compensating transaction.

Gro := GRO *C*;
Commit-Seq := CRO STATE(*Gro, commit*);
Abort-Seq := CRO STATE(*Gro, abort*);
($\forall i$, GET(*Abort-Seq*, i) *BA* GET(*Abort-Seq*, $i+1$)) & LAST (*Abort-Seq*) *BA* FIRST(*Commit-Seq*) & (NUM(*Commit-Seq*) = 1)

The functionality of operator STATE is similar to that of INVOKER in Example 2. It groups all job steps in a group by their states. Thus, *Commit-Seq* (*Abort-Seq*) is the sequence by invocation order of all committed (aborted) job steps in group *Gro*. Operation NUM returns the total number of the elements in the sequence. The 4th statement is interpreted as: C is initiated only after its previous initiation has aborted (represented by the two BA dependencies) and must commit exactly once (represented by operator NUM).

4 C-unit

4.1 Concepts and Definitions

As mentioned before, our model is intended to be used for those applications where some integrity constraints on data exist across the boundary of sites. These cross-boundary constraints may impose stringent consistency requirements relating to a collection of job steps. For this kind of collections of job steps, the correctness of the execution may be compromised by the improper interleaving[2] To cope with this problem, we must impose certain kind of transactional properties on the collections. We call a collection of this kind a *C-unit*. (Stands for "consistency unit".) Note that the purpose of forming a C-unit is purely for accommodating the integrity constraints, not for achieving any other semantically meaningful goal at the conceptual level. In the following, for easy presentation, if for consistency reason several job steps must be included into a transaction with traditional ACID properties, we say that they are tied by ACID.

A C-unit is formed inside a workflow. Its execution must be part of the workflow. In other words, except for being executed as a single consistency unit, all the job steps in the C-unit must be executed according to the internal structure of the corresponding workflow. Formally, we have

Definition 1: Let $W =< S, D >$ be a workflow where $S = \{s_1, \cdots, s_n\}$ is the set of job steps and $D = \{D_1, \cdots, D_m\}$ is the set of dependencies. A C-unit for W is a pair $X =< Y, Z >$ where $Y \subseteq S$ and $Z = \{aRb : aRb \in D \ \& \ a \in Y \ \& \ b \in Y\}$, where aRb implies b has a dependency R on a.

In the definition, we call W the encompassing workflow of X. Thus, dependencies in a C-unit must be compatible with those in the encompassing workflow. In addition, if two members of a C-unit are related by certain type of dependency for the workflow, they must also be related by the same type of dependency for the C-unit. Thus the dependencies for the C-unit are completely determined by those for the encompassing workflow.

To determine the membership of a C-unit, two criteria should be taken into account: correctness and efficiency. For correctness reasons, a C-unit must contain *all* job steps which are tied by ACID. For efficiency reasons, a C-unit should contain *only* the job steps which are tied by ACID. The implication here is that any job step, with the exception of compensating job steps (explained below), that is not tied with the other members of a C-unit by ACID should not participate in that C-unit.

If a job step belongs to a C-unit, we require that its compensating job step, if any, also belongs to the C-unit. The reason is twofold. First, this treatment lets us avoid using a concept like "compensating C-unit" to handle the failure of a C-unit, and hence simplifies the model. (Refer to Section 3.3.3.) Secondly, as will become clear in Section 3.3.4, compensating job steps will not cause any additional delay for the containing C-units to release the resources, and therefore their inclusion into a C-unit does not have a negative impact on the performance.

By the definition, a compensable member of a C-unit is allowed to commit as soon as it finishes the execution, i.e., without waiting for the commit of the C-unit itself. However, after the commitment it still must hold the resources

[2]We are aware of the problems caused by the LT-autonomy of local databases; no global integrity constraint can be enforced on the global level unless the inter-site correctness problem is properly addressed [23].

until the C-unit terminates, since otherwise improper interleaving may happen in case of concurrent execution of multiple workflows. Although this is not desirable in terms of performance, it is necessary to ensure the correct execution of workflows.

Unlike a compensating job step, an alternative of a member of a C-unit may or may not belong to that C-unit. Whether or not a job step and its alternative will both be the members of a C-unit depends on if both of them are tied by ACID to some *other members* of this C-unit[3].

Since a C-unit is essentially a group of job steps, it can be specified using the techniques in Section 3.2.2. For example, if the first job step of sequence *seq1* and the second job step of sequence *seq2* form a C-unit, we write it as {GET(*seq1*, 1), GET(*seq2*, 2)}. If for all i, the ith job step in *seq1* and the ith job step in *seq2* form a C-unit, then we express this by $\forall i$, {GET(*seq1*, i), GET(*seq2*, i)}, meaning for all legal ordinal number i in the sequence, this pair is a C-unit.

Example 5: In PortNet system, whenever the agent initiates *PRE-AAN-AGENT*, message AAN will be sent to the port and NBN to initiate *PRO-AAN-PORT* and *PRE-AAN-NBN*, respectively. The initiations result in three job steps storing the same information into the corresponding databases. We require that the information in the three databases be consistent. To ensure this, one approach is to group the related job steps into a C-unit, as follows.

Seq1 := CRO (INVOKER(*PRE-AAN-AGENT*, *Agent*);
Seq2 := CRO (INVOKER(*PRO-AAN-PORT*, *Agent*);
Seq3 := CRO (INVOKER(*PRO-AAN-NBN*, *Agent*);
$\forall i$, {GET(*Seq1*, i), GET(*Seq2*, i), GET(*Seq3*, i)};

Note that in the first three statements, we create sequences of those job steps generated from the invocations only by the agent, and leave out those generated by the appointed person.

In general, there may be more than one C-unit in a single workflow, and two different C-units in the same workflow may or may not overlap. However, no C-unit should be a proper subset of the other. The reason is clear: if a C-unit is a proper subset of another C-unit, then for efficiency reason the difference between the latter and the former should not be included in the latter (Refer to the discussion earlier in this subsection.)

4.2 Committing a C-unit

It may seem that we can define commit and fail of a C-unit in a similar fashion to that for the workflow, i.e., in terms of acceptable states of a C-unit. Before we give a definite answer, we first consider an example. Suppose a workflow is defined as $W = (S, D)$ where $S = \{s_1, c_1, s_2, c_2, s_3, c_3\}$, where c_i is the compensating job step of s_i, and s_3 is the alternative of s_2. We further assume that s_1 is tied by ACID properties to s_2 but not to s_3. According to the way a C-unit is formed, We have $C = (S_1, D_1)$ where $S_1 = \{s_1, c_1, s_2, c_2\}$. Consider the following scenario: s_1 commits, s_2 aborts, and s_3 commits. Since s_3 is functionally equivalent to s_2, the fact that s_3 commits implies that the goal of s_2 has been achieved. Therefore both the goal of s_1 and that of s_2 have been successfully achieved. The intuition here is that C should be considered to commit, and therefore state (C,N,A,N) should be an acceptable state for C. Now consider another scenario: s_1 commits, s_2 and s_3 abort. Now the goal of s_2 has failed to be realised, resulting in a failure of the workflow. This suggests that state (C,N,A,N) be treated as an unacceptable state. The contradiction implies that a commit or a failure of a C-unit should not be defined based only on the states of its members. This leads to the following

Definition 2: A C-unit commits if for each member s, if s is a compensating job step then it has not been started, otherwise either s commits, or it has exactly one alternative that commits. Otherwise, we say that it fails.

Notice that in the latter condition, the alternative that commits is not necessarily a member of the C-unit.

We observe that the only practical significance of the commit of a C-unit is that it can release the resources it holds for its compensable members. In other words, the commitment of a C-unit does not invoke complicated I/O operations for effect-installation.

In practice, an application designer has the flexibility to restrict or relax the correctness requirements of a C-unit. For example, if the application requires that the cross-boundary constraints be rigorously maintained, then a strong atomicity or isolation may be required as the properties of a C-unit. On the other hand, if the application makes use of many abstract operations, then probably semantic serialisability can be used to take advantage of the rich semantics of these operations in generating the correct interleaving.

4.3 Dissolving a C-unit

When a C-unit fails, meaning that some of its members fail to achieve their goal, we may retry the execution of the failed members until it is successful. In the meantime, all the resources held by the C-unit are unavailable to other job steps of (the same or different) workflows. If we give up the retry, then the C-unit must be compensated for. As mentioned before, compensating a C-unit does not require a "compensating C-unit", since any compensable member is associated with a compensating member. In other words, to compensate for a C-unit, each of its members must be compensated/rolled back individually, without the concern of the existence of the C-unit. This implies that at this time the C-unit should cease to exist, as far as recovery is concerned. If a C-unit ceases to exist, we say that it has *dissolved* itself. When a C-unit dissolves itself, its previous members now gain the same properties as those possessed by other job steps which never belong to any C-unit. In particular, if any of them is compensable and has committed or aborted, then it can release the resources it holds.

A practical consideration is when a C-unit can dissolve itself without causing inconsistent execution. In the following, to simplify the presentation, we will say that a C-unit *can dissolve itself* if such a dissolving does not cause inconsistency. In general, a C-unit can dissolve itself if it has committed or failed (full isolation). In the next section, we will see that in some cases, a C-unit can dissolve itself before its final status is known.

4.4 Coping with C-units as a long running activity

Since it contains a group of job steps, a C-unit may itself be a long running activity. If a long running C-unit prevents other workflows from accessing the resources it is holding, then the significance of our arranging the original activity into a workflow diminishes.

Several approaches can be used to alleviate the problem caused by long running C-units. The first is to rely on the application. If the job steps and local transactions are semantically rich, then it is possible that at a particular site

[3]Interleaving between a job step and any of its alternatives is not a concern, since if one commits, the other will be idle.

job steps and local transactions or job steps of different C-units (in different workflows) are commutable. When this happens, a job step in a C-unit may release the resources after it commits.

The idea behind the second approach is that sometime a C-unit can dissolve itself before its final status (commit/abort) has been determined. When this happens, those C-units should be explicitly identified in the model. We first look at an example of when this would happen. Consider the C-unit in the example of Section 3.3.2. For easy reference, we rewrite it here. $W = (S, D)$ where $S = \{s_1, c_1, s_2, c_2, s_3, c_3\}$, and $C = (S_1, D_1)$ where $S_1 = \{s_1, c_1, s_2, c_2\}$. We assume that s_2 and s_3 are functionally equivalent. We further assume that s_1 and s_2 are tied by ACID but s_1 and s_3 are not [4]. Suppose s_1 commits but s_2 aborts. Now whether or not C can commit depends on whether or not s_3 will commit. We argue that if s_3 is initiated as a result of the abort of s_2, it is possible for C to release its resource without waiting for the termination of s_3. The rationale is the following. Since s_3 is not tied by ACID to any members of C, any interleaving between C as a whole and s_3 is not a problem even if s_3 later commits. On the other hand, if s_3 eventually aborts, then C fails. If the retry of any of s_2 and s_1 is not attempted, then the earlier release of the resources by s_1 also will not lead to incorrect execution, since all the effects of s_1 will be compensated anyway. In general, we have

Assertion 1: Let $C = (X, Y)$ be a C-unit where $X = X_1 \cup X_2$, where $X_1 = \{s : s$ is a compensating job step$\}$ and $X_2 = \{s : s$ is not a compensating job step$\}$. We further write X_2 as $X_2 = (\{s_1\} \cup U_1) \cup, \cdots, \cup (\{s_n\} \cup U_n)$ where each member of U_i is an alternative of s_i. If the following conditions are met, and no retry will be attempted for any aborted members, if any, of C, then C can dissolve itself:

1. some member of X_1 has been started, or
2. for all $1 \leq i \leq n$, either one of the members of $\{s_i\} \cup U_i$ commits, or all members of $\{s_i\} \cup U_i$ abort.

Note that the no retry clause in the assertion implies that if C fails, then all its committed compensable members will be compensated for.

Idea of the proof: Assume the first condition is true. Note that starting a compensating job step implies the job step to be compensated for has committed, but its effects have to be (semantically) wiped out. This means that C currently has failed and therefore all its committed compensable members will be compensated for. Thus releasing the resources held by those members is allowed since their (direct or indirect) effects will be compensated by the corresponding compensating job steps.

Now assume the first condition is false but the second condition is true. Let $L = \{s_k : 1 \leq k \leq n$ and all members of $\{s_k\} \cup U_k$ abort$\}$. If $L = \phi$, then for all $1 \leq i \leq n$, one of the members of $\{s_i\} \cup U_i$ commits. This means C commits. Thus C can dissolve itself. Now assume $L \neq \phi$. If there is a $s_i \in L$ such that all its alternatives belong to C, then U_i is the set of these alternatives. This implies that C has failed. Since no retry is attempted for any aborted members of C, all the committed compensable members of C will be compensated for. Thus the last statement in the last paragraph applies. Now suppose for all $s \in L$, s has some alternative which does not belong to C. Let $M = \{V : s \in L$ and V is the set of alternatives of s which do not belong to $C\}$. If for all $V \in M$, there is a $r \in V$ such that r will commit, then C will commit. Since r is not tied by ACID to any member of C, interleaving between r and the members of C is allowed. Thus the claim in the assertion is true. If, on the other hand, for some $V \in M$, all $r \in V$ will abort, then C will fail. In this case, similar arguments to those earlier in this paragraph can establish the claim in the assertion. Q.E.D.

Note that one fact implied by this assertion is that a C-unit never has to wait for compensating job steps to commit before it dissolves itself. Thus, the inclusion of compensating job steps into C-units does not delay the release of the resources held by the C-unit.

Whether or not the C-unit satisfying the conditions in the above assertion will actually be dissolved depends on the choice of the application. In particular, if the designer decides that should C eventually fail, any of its aborted members would not be retried, meaning the current execution of C will be given up, then C *should* be dissolved immediately once the two conditions in the assertion become true.

4.5 Consistency of workflows

Due to the presence of C-units, which may span multiple sites, the consistency of a workflow is not as obvious as other models where individual job steps are independent units. We discuss this issue in two cases.

When a workflow is executed alone, then its correct execution can be ensured through the correct enforcement of the dependencies among its job steps. If the workflow contains no C-unit, or if all C-units are disjoint, the concurrent execution of the job steps or C-units in a single workflow posts no problem. If there are overlapped C-units, then the concurrent execution of these C-units must in some sense be equivalent to a serial execution of them. This is guaranteed by the (relaxed) isolation property we require of C-units.

When multiple workflows are executed concurrently, individual job steps from different workflows may be processed at the same sites. However, since all job steps which are related to each other by ACID have been grouped into respective C-units, C-units and job steps which do not belong to them can be viewed as independent entities. Since it is guaranteed that in a concurrent execution of multiple workflows, any C-unit will behave as if it is executed alone, the conclusion that any workflow is consistent even in concurrent execution is reasonable. The tricky part is to specify C-units in such a way that cycles in the global serialisation graph do not occur.

4.6 Failure Recovery

Since many workflows are long running activities in nature, we would like them to be forward recoverable. Forward recovery is made possible by such facilities as retry, alternative, check-pointing, etc. For those workflows in which all job steps are independent computational units, this approach is usually feasible. However, if a workflow contains C-units, then the job steps in a C-unit are tied by ACID. In this case, due to the presence of local transactions, it may not always be possible to define alternatives for them. Similarly, it may not always be correct to rerun a job step in a C-unit. In other words, a C-unit itself may not be forward recoverable.

Whether or not a C-unit is forward recoverable depends on the characteristics of the value dependencies pertaining to that C-unit, and the facilities provided by the local processing unit, as well as the extent to which the local auton-

[4] We are not concerned with the ACID-coupling of s_2 and s_3. See the footnote of Section 3.3.1.

omy can be compromised by the local processing unit. In general, if the value dependency between the job steps in a C-unit does not form a cycle, or if it is possible to limit the accessibility of local transactions during the recovery period, then forward recovery of the C-unit is possible [19, 24, 23].

In case where a C-unit is not forward recoverable, backward recovery must be used to eliminate the effects of all its members. As mentioned in Section 3.3, this will be done on the individual job step basis automatically by the facilities provided by the workflow. For compensable job steps of a C-unit, the corresponding compensating job steps will be executed. For non-compensable job steps, abort operations will be performed at the local level. This requires local autonomy to be restricted.

5 System support outline

The goals of the system support are, first, ensuring the dependencies between job steps are correctly enforced, and second, ensuring the improper interleaving does not occur in the concurrent execution of workflows.

In [2], a system architecture model is proposed where two basic components, workflow manager and transaction manager each takes the responsibility of one of the goals stated above. Specifically, workflow manager must guarantee the execution of job steps does not contradict the inter-job step dependencies prescribed in the application specification while the transaction manager provides required isolation.

While this general framework does apply to the system support for our model, special features must be incorporated to accommodate the extended generality provided by the model. Firstly, since the job step in our model is not explicitly named in the dependency specification, the system must contain necessary component that can unambiguous identify the job steps in each dependency specification. In addition, our model allows dependencies to be specified on the group basis. Thus whenever a job step is generated, the groups it belongs that participate in specified dependency must be correctly located. Secondly, mechanisms are needed to differentiate C-units that contain different job steps generated by the same step plan.

To support these special features, we include two more modules, *job step locator* and *C-unit identifier*, into the system support. Figure 3 is a sketch of the system architecture.

The job step locator accepts the definition of each sequence and group. When a job step is submitted, the job step locator analyses its related information, such as invoker, invocation time, etc. It then inserts the job step into the groups and/or the sequences it belongs. It converts the dependency specifiers into the dependency specifications on individual job steps which involve the job step being submitted. This conversion is achieved by first searching for all operations in any dependency specifier which are defined on a sequence or group the job step belongs, and then executes those operations. In cases where the values necessary for locating a job step is not known at the time of submitting, the job step locator must make the most conservative choice. For example, when job step $s \in G$ is being submitted, the dependency specifier STATE(G,commit) BC GET(Seq,1) will be converted to s BC r where r =GET(Seq,1), even if it is not known at the time of submission whether or not s will commit. The converted dependency specification will be accessed by workflow dependency manager.

The workflow dependency manager determines if the job step can be accepted by consulting the dependency specification produced by the job step locator. The determination process usually involves complicated rules [1, 20]. In general, it makes a positive decision only if it is certain that accepting the job step does not lead to a future situation where there is no choice but to violate some dependency.

The membership information about the submitted job step will also be used by the C-unit identifier. Using this information together with the C-unit definition contained in the application specification, the C-unit identifier will execute the operations on the relevant sequences or the groups to determine which C-unit the job step belongs. It then passes this information to the transaction manager for ensuring the required isolation.

As we mentioned before, in the presence of a C-unit, forward recovery and backward recovery may both be needed. A problem is who should be in charge of what. In our opinion, the workflow manager can still take the charge of forward recovery. This involves the correct check-pointing and the arrangement for the initiation of compensating job steps, re-submission of the aborted job steps, running alternative job steps, etc. The workflow manager should also take care of backward recovery, if any, for those parts of the workflow which do not belong to any C-unit. On the other hand, it is noted in [2] that the transaction manager should be responsible for the backward recovery for any C-unit, since this can be readily incorporated into its concurrency control policies, and therefore can achieve better efficiency.

6 Conclusion

In this paper, we study the modelling aspects for workflow applications which have more general requirements than those modelled by many existing models. In such applications, the job steps cannot be determined in advance, and there are dependencies between job steps generated by the same program. We propose to incorporate more semantic components into the model, namely, sequences and groups of job steps. We illustrate how these types can be used to cope with the dependency specification for this kind of applications.

We also study C-unit, which may be a necessary component for any workflow systems where integrity constraints span multiple processing units. We discuss a number of related issues for C-units, such as its properties, and impact on the workflow in terms of efficiency, and approaches to alleviate the problem caused by long running C-units. We also discuss several essential issues about the system support.

Several things are for further study, e.g. how dynamically the C-units can be specified and what is required from a a distributed system support in terms of protocols etc.

Acknowledgements The authors wish to thank anonymous referees for many helpful comments and suggestions. We also thank the TransCoop teams in Darmstadt and Enschede for many inspiring discussions on the topic. Special thanks go to Juha Puustjärvi and Henry Tirri for valuable comments in the final preparation phase of the paper.

References

[1] M. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *The 19th International Conference on VLDB*, 1993.

[2] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *Sigmod Record*, 22(3), September 1993.

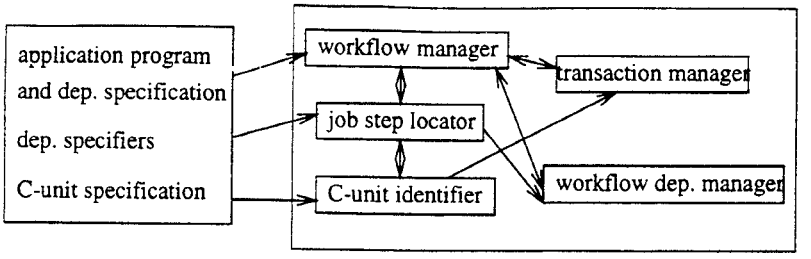


Figure 3. A sketch of system architecture.

[3] O. Bukhres, A. Elmagarmid, and E. Kuhn. Implementation of the flex transaction model. *IEEE Data Engineering Bulletin*, 16(2), June 1993.

[4] P. Chrysanthis and K. Ramamritham. A formalism for extended transaction model. In *The 17th International Conference on VLDB*, 1991.

[5] Workflow Management Coalition. Workflow management coalition, june 1995 meeting. Oral communication, 1995.

[6] Workflow Management Coalition. Workflow management coalition overview, 1995.

[7] U. Dayal, M. Hsu, and R. Ladin. A transaction model for long-running activities. In *The 17th International Conference on VLDB*, 1991.

[8] M. Hsu (ed.). Special issue on workflow and extented transaction systems. *Bulletin of TC on Data Engineering, IEEE CS*, 16(2), June 1993.

[9] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multibase transaction model for interbase. In *The 16th International Conference on VLDB*, 1990.

[10] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

[11] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Modeling long-running activities as nested Sagas. *IEEE Data Engineering Bulletin*, 14(1), March 1991.

[12] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *The 10th International Conference on Data Engineering*, pages 462–473. IEEE Computer Society, February 1994.

[13] D. Georgakopoulos, M. Hornick, F. Manola, M. Brodie, S. Heiler, F. Nayeri, and B. Hurwitz. An extended transaction environment for workflows in distributed object computing. *IEEE Data Engineering Bulletin*, 16(2):24–27, june 1993.

[14] D. Georgakopoulos, M Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automaton infrastructure. *Distributed and Parallel Databases, An International Journal*, 2(3), Sept. 1994.

[15] J. Grudin. Computer supported cooperative work: History and focus. *IEEE Computer*, pages 19–27, May 1994.

[16] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2), 1994.

[17] D. McCarthy and S. Sarin. Workflow and Transactions in InConcert. In *[8]*.

[18] M. Nodine, S. Ramaswamy, and S. Zdonik. A cooperative transaction model for design databases. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 3. Morgan Kaufmann Publishers, 1992.

[19] J Tang and Y. Sun. Coping with value dependency for failure recovery in multidatabase systems. In *Proc. of the 4th Int'l. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*. IEEE CS, May 1994.

[20] J. Tang and J. Veijalainen. Enforcing inter-task dependencies in transactional workflows. In *Proc. of the 3rd Intl. Conf. on Cooperative Information Systems*. CoopIS, May 1995.

[21] T. Tesch and P. Verkoulen (eds). Requirements for the cooperative transaction model. TransCoop Deliverable II.1; TC Technical Report, GMD,University of Twente, VTT, January 1995.

[22] J. Veijalainen. *Transaction Concepts in Autonomous Database Environments*. R.Oldenbourg Verlag/GMD, 1990.

[23] J. Veijalainen. Heterogeneous multilevel transaction management with multiple subtransactions. In *Fourth Intl. Conference on Database and Expert Systems Applications, DEXA '93, LNCS Nr. 720*, pages 181–188. Springer-Verlag, Sept. 1993.

[24] A. Zhang and J. Jing. On structural features of global transactions in multidatabase systems. In *Proc. of the 3rd Workshop on RIDE*. IEEE CS, Feb. 1993.