Correctness of Workflows in the Presence of Concurrency *

Ismailcem Budak Arpinar[1], Sena (Nural) Arpinar[1], Ugur Halici[2], and Asuman Dogac[1]
Software Research and Development Center
[1]Dept. of Computer Engineering
[2]Dept. of Electrical Engineering
Middle East Technical University (METU)
06531 Ankara Turkiye
{budak, nural, asuman}@srdc.metu.edu.tr, halici@rorqual.cc.metu.edu.tr

Abstract

Workflow processes are long-duration activities and therefore it is not possible to apply the well accepted correctness techniques of transactions directly to workflow systems.

In this paper, we first mention the correctness problems of workflow systems and then exploit the available semantics in workflow specification in the form of data and serial control-flow dependencies to define isolation units. We show that isolation units in a workflow can be identified automatically, i.e. without human intervention, from the workflow definition. We then propose a technique to provide for the correctness of concurrently executing workflows on the basis of isolation units. The technique is general enough to handle the correctness of hierarchically structured workflows consisting of compound tasks.

Keywords: *Workflow System, Correctness of Workflow Systems, Concurrency Control in Workflow Systems, Isolation Units.*

1 Introduction

A workflow consists of a set of processing steps (tasks) together with some specification of the control and data-flow between these tasks. Although there is some work on the interactions among concurrently executing workflows, the issue has not been completely resolved yet. As long as we deal with loosely coupled systems where no integrity constraints exist that span multiple systems, the single tasks of a workflow can be executed without any further control. In a more tightly coupled system, however, there are dependencies that must be observed. The conventional techniques used in concurrency control are not suitable for workflow environments because workflow execution may take several days or weeks.

In this paper, we first mention the correctness problems in workflow systems. Then by using the data and serial control-flow dependency information in the workflow definition, we introduce isolation units, i.e. the parts of a workflow that must be executed in synchronization to provide correctness. We show that isolation units can be automatically identified within a workflow system. We develop a technique based on isolation units which allows for correct execution of concurrently executing hierarchically structured workflows consisting of compound tasks.

The paper is organized as follows: Section 2 presents the related work. In Section 3 our basic workflow model and correctness problems in workflows are explained. Section 4 introduces isolation units and correctness of nested tasks of workflows. In Section 5, NT (Nested Tickets) technique for the correctness of concurrently executing workflows is presented. We conclude with Section 6.

2 Related Work

2.1 Invariants of ConTract Model

In the ConTract model [WR 92] in order for tasks to work correctly, predicates named as invariants are defined to hold on the database. Invariants do not solve the problem of improper interleaving of two or more tasks from different workflows at multiple sites. In [WR 92], authors state that in many cases it is sufficient to make sure that a certain tuple is not deleted; that a certain attribute value stays within a specified range; that there are no more than a certain number of certain type of tuples, etc. to ensure correct execution of workflows and the workflow designer can specify these constraints as invariants.

For example, consider the two tasks of a "Business

* This work is partially being supported by the Turkish State Planning Organization, Project Number: AFP-03-12DPT.95K120500, by the Scientific and Technical Research Council of Turkey, Project Number: EEEAG-Yazilim5, by Motorola (USA) and by Sevgi Holding (Turkey)

Trip Reservations Workflow" named as *Travel_Data_Input* and *Flight_Reservation*. Exit invariant of first task is specified as ($budget > cost\_limit$) and entry invariant of second task is specified as (($budget > cost\_limit) and (cost\_limit > ticket\_price)$). At the execution time, the run-time system checks at the end of execution of a task if the predicates are valid. If they are valid, the constraint is satisfied and the transaction which protects the step is allowed to commit. After this task is committed, other tasks of concurrent workflows can access and update the variable in the predicate which resides in a shared Resource Manager (RM). However, when a second task starts its entry invariant is evaluated to ensure correct execution. For example, after *Travel_Data_Input* task accessed *budget* and committed, other tasks of concurrent workflows can update *budget*. In *Flight_Reservation* task *budget* is accessed and its entry invariant (($budget > cost\_limit) and (cost\_limit > ticket\_price)$) is evaluated. If it evaluates to false, the task is not allowed to start. So ConTracts permits unserializable executions but enforce application specific correctness.

Alternatives for predicate specification can be a state based approach, CNF (Conjunctive Normal Form), or first-order logic expressions (CNF plus quantifiers) or a more powerfull method. Yet it may be difficult to determine and/or to enforce the invariants.

2.2 Step Compatibility

In [BDS 93] to ensure data consistency, semantic serializability of workflows is proposed as the correctness criterion. A human expert declares a compatibility matrix for tasks of a workflow. Compatibility of two tasks means that the ordering of two tasks in a schedule is insignificant from an application point of view. If two tasks are not defined as compatible they are in conflict. A schedule is semantically serializable if an equivalent serial execution exists with the same ordering of conflicting tasks. For example *Risk_Evaluation* and *Risk_Update* tasks of different Loan Request Processing workflows can be defined as in conflict whereas two *Enter_Decision* tasks of different workflows can be defined as compatible although two *Enter_Decision* tasks update the same data item. Hence *Risk_Evaluation* and *Risk_Update* tasks of different workflows must be executed serializable to ensure the consistency of banks total involvement.

In [BDS 93], the compatibility matrix is restricted to the tasks of different instances of the same workflow type, e.g. compatibility matrix for the tasks of two Loan Request Processing workflows is defined. But in real applications tasks of different workflow types can be executed concurrently and a compatibility matrix should be defined for them, for example, between the tasks of a Loan Request Processing workflow and tasks of a Risk Management workflow.

2.3 Transaction Specification and Management Environment (TSME)

In TSME [GHM 95] using the transaction specification language, correctness as well as state dependencies can be specified between tasks of workflows. Different correctness dependencies such as serializability, temporal, cooperative dependencies can be specified. For example for the concurrent execution of two alternative line provisioning tasks of a Provisioning and Billing workflow for a telecommunication application the correctness criteria can be specified as serializability; or if one of them is allowed to commit they may use same lines and slots and the correctness criteria can be specified as cooperative.

To define conflicts, each objects is associated with a conflict table. Serialization dependencies are specified as acyclic serialization order dependencies between tasks. Temporal order dependencies are specified by giving specific serialization order between tasks. Cooperation between tasks is provided by using breakpoints or augmenting conflict tables of shared objects. Two cooperating tasks read and write specific objects without restrictions at breakpoints or some tasks are defined as non-conflicting on specific objects.

2.4 M-serializability

In [RS 94], M-serializability is defined as a correctness criterion for concurrent execution of workflows. In this model, related tasks of a workflow are grouped into execution atomic units. M-serializability requires that tasks belonging to the same execution atomic unit of a workflow have compatible serialization orders at all sites they access. Yet this approach does not consider the nesting of tasks.

2.5 Multilevel Atomicity

In [L 83], transactions are grouped into semantic types and a transaction can belong to more than one semantic type. Each type has different sets of breakpoints, inserted between the steps of a transaction at appropriate points. Steps of compatible transactions can be interleaved at these breakpoints. This idea can be adopted to workflows by inserting appropriate breakpoints between tasks of a workflow, but due to autonomy of local sites intervention of local transactions can not be restricted by breakpoints. Commitment of individual tasks are breakpoints from the viewpoint of local transactions.

2.6 Commercial and Prototype WFMSs

Most commercial WFMSs provide limited capabilities for concurrency control. XAIT's InConcert [DS 93] supports a form of check-in and check-out model which is

a primitive way for concurrency control. Lotus Notes [GHS 95] allows a user to update an object and create a new version of it. When very large amount of objects are updated, this method is not feasible because keeping every version of an updated object is very costly. Staffware [GHS 95] uses a pass-by-reference/pass-by-value approach for concurrency control. Data items that can be shared among multiple clients are passed by reference, i.e. clients access a centrally stored data item using a pointer, possibly concurrently. Mentor [WWW 96] supports the distributed execution of workflows and uses a TP Monitor, namely Tuxedo to provide atomicity of distributed transactions. The synchronization is provided by means of update messages between workflows at synchronization points. The ATM [DHL 91] approach includes an extended nested transaction model and language for describing long running activities.

2.7 Multilevel Transaction Frameworks

Workflows may contain a hierarchy of tasks. Therefore we need to deal with the concurrent execution of nested tasks. In [BBG 89] a concurrency theory is provided for nested transaction systems. In this theory, commutativity and pruning concepts are used to prove the correctness of a concurrency control technique. Principles and realization strategies of multilevel transaction management is described in [W 91]. In [HAD 97] we have developed a theory for the serializability of nested transactions in multidatabases.

3 Workflow Correctness Issues

3.1 Workflow Model

In this section we define the basic workflow model to be used throughout the paper.

The individual steps that compromise a workflow are termed as *tasks*. Tasks may involve humans as well as programs. Each task has a set of input and output parameters. A task includes Data Manipulation (DM) operations or subtasks. Hence, a workflow is a tree of tasks, the subtrees of which are either nested or flat tasks.

A Workflow Management System (WFMS) involves distributed objects managed by either a number of pre-existing and autonomous Local DBMSs (LDBMSs) (e.g. Sybase[1], Adabas D[2]), or non- transactional Resource Managers (RMs) (e.g. file systems) as well as human participants. These LDBMSs, non-transactional RMs, and human participants may exist on a distributed heterogeneous platform.

[1]Sybase is a trademark of Sybase Corp.
[2]Adabas D is a trademark of Software AG Corp.

In a WFMS environment there exists at least three types of tasks and transactions:

- **Local Transactions**, those transactions that access data managed by a single DBMS and they are executed by the LDBMS, outside the control of WFMS,
- **Transactional Tasks**, those tasks that are executed under WFMS control and they access data controlled by RMs with transactional properties (i.e. ACID). Transactional RMs offer at least two transactional operations: commit and abort,
- **Non-Transactional Tasks** are also executed under WFMS control, but they access data controlled by RMs without transactional properties such as file systems. Yet it is possible to introduce transactional properties to these systems, for example by wrapping non-transactional RMs to provide transaction and concurrency control services according to OMG's Object Transaction Service (OTS) [OMG 94] and Concurrency Control Service (CCS) specifications. Hence these RMs can behave similar to transactional RMs.

From this point on *task* and *transaction* will be used interchangeably throughout the paper. Both of the terms refer to an atomic unit of work in general.

A task or a local transaction t_i is a sequence of read (r_i) and write (w_i) operations terminated by either a commit (c_i) or an abort (a_i) operation from the concurrency control perspective. A single task may contain Data Manipulation (DM) operations at more than one site. Note that DM operations are invisible to the workflow system.

There are two types of flow dependencies between tasks of a workflow:

- **Data Flow Dependencies** map an output parameter of a task to input parameter of one or more tasks.
- **Control Flow Dependencies** specify the execution dependency between the tasks.

3.2 Correctness Problems in Workflow Systems

The two correctness problems arising from the concurrent execution of tasks in workflow systems are discussed in the following through examples. In these examples we choose to explicitly show the DM operations although they are not visible at the workflow level, just to provide clarification to the problems presented.

Example 1. Consider two concurrently executing Airline Reservation workflows as shown in Figure 1. A

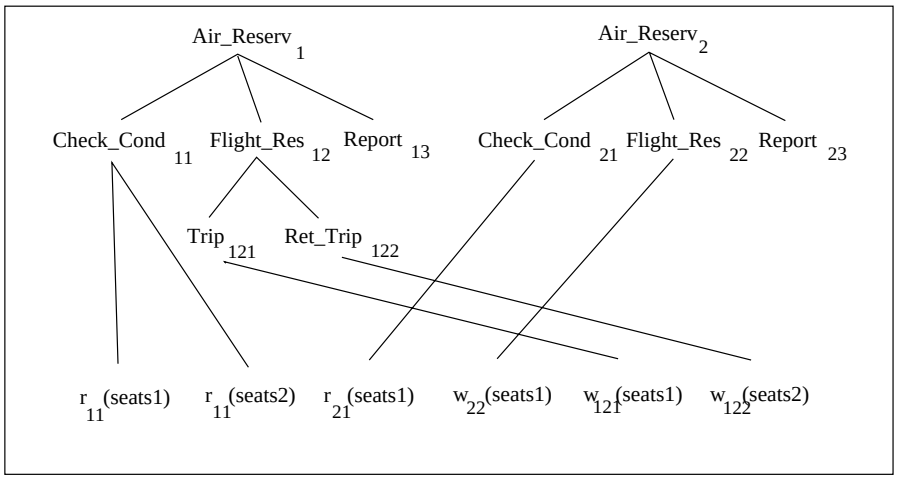

Fig. 1. Concurrency Control Problem of Workflows At A Single Site

customer wants to make a round trip flight from Istanbul to Paris. Therefore, an Airline Reservation workflow (Air_Reserv_1) is created. Check Condition task ($Check\_Cond_{11}$) of Air_Reserv_1 workflow checks the available *seats* for both Istanbul to Paris and Paris to Istanbul flights from the Flight Reservation Database. If there are available seats in these flights, Flight Reservation task ($Flight\_Res_{12}$) is started. $Flight\_Res_{12}$ task is broken into two subtasks, Trip ($Trip_{121}$) which reserves the flight from Istanbul to Paris, and Return Trip (Ret_Trip_{122}) which reserves the flight from Paris to Istanbul. Report task ($Report_{13}$) writes flight information of the customer to her ticket. Air_Reserv_2 is another instance of the same workflow and also updates available seats for Istanbul to Paris flight ($seats1$).

In Example 1 the problem arises because after reading $seats1$ and $seats2$, $Check\_Cond_{11}$ commits and $Flight\_Res_{22}$ task of Air_Reserv_2 updates $seats1$. However, the previously read value of $seats1$ by $Check\_Cond_{11}$ is used later in processing of Air_Reserv_1 workflow to control the flow of task $Flight\_Res_{12}$. Note that this value of data is no longer valid. □

This example considers correctness problem of data residing on a single site. We would like to point out that passing references instead of data itself, does not solve the inconsistent data flow problem since a task may perform a certain computation to create the data to be used by another task. In this case, even if the data is stored to be accessed through a pointer, it may no longer be correct because the underlying data used in computing this data may have changed.

Next example demonstrates that data consistency can be violated by the concurrently executing workflows at multiple sites.

Example 2. Consider the two workflows in Figure 2. $Transfer_1$ workflow transfers money from one account to another. These accounts are in different subsidiaries of a bank (i.e. different sites). $Transfer_1$ workflow includes two tasks, namely $Withdraw_{11}$ and $Deposit_{12}$. $Withdraw_{11}$ task withdraws the given amount of money from $acc1$ at the first site by means of read and write operations on the underlying records as shown in Figure 2. $Deposit_{12}$ adds the given amount of money to $acc2$ at the other site. $Audit_2$ workflow checks the balance of the bank by summing up all the accounts in the bank's subsidiaries in $TotalBal_{21}$ task. $TotalBal_{21}$ has two $Check$ subtasks for $Site_1$ and $Site_2$. Balances accessed by $Check_{211}$ and $Check_{212}$ are summed in Sum_{213}. Also assume each workflow executed in the system updates a log record for bank's security and statistical purposes. $Report_{13}$ and $Report_{22}$ tasks update this log record which is located at $Site_3$.

The schedule in Figure 2 is not correct because $Audit_2$ sees an inconsistent result, since it misses the money being transferred from $acc1$ to $acc2$. In order to prevent this inconsistency, the tasks of $Transfer_1$ and $Audit_2$ workflows must be executed so that they have compatible serialization orders at each site. □

4 Concurrency Control for Workflows

Data consistency can be violated by improper interleaving of concurrently executing workflows as shown in Examples 1 and 2. Also, such inconsistencies can occur due to improper interleaving of concurrently executing workflows and local transactions. Such interleavings must be prevented to ensure data consistency

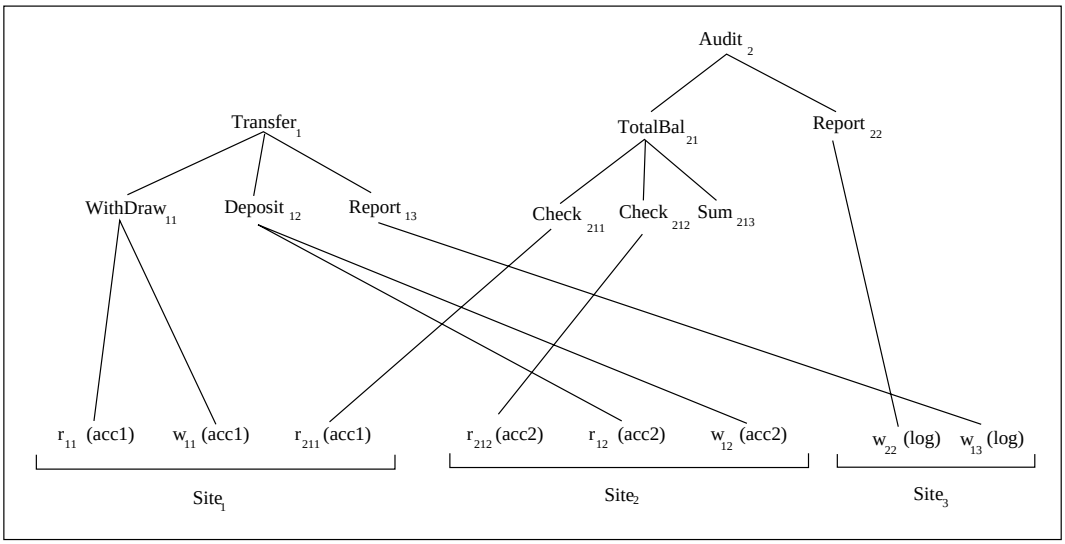

Fig. 2. Concurrency Control Problem of Workflows At Multiple Sites

in WFMSs. In this section we introduce the "isolation unit" concept and a related technique to provide for the correctness of concurrently executing workflows. In achieving this goal we aim at increasing concurrency. Our starting point is to exploit the available semantics in workflow specification. How this semantic knowledge is extracted by using *data* and *serial control-flow* dependencies between tasks is discussed in Section 4.1. Usage of this knowledge to preserve data consistency is provided in Section 4.2.

4.1 Isolation Units

We define an isolation unit to be the set of (sub)tasks that have data-flow and also serial control-flow dependencies among them. We claim that the workflow correctness can be provided by identifying the isolation units in a workflow system automatically from the data and serial control-flow dependency information obtained from the workflow specification. Before providing a formal definition of an isolation unit we will provide some motivating examples.

Consider Example 1. $Check\_Cond_{11}$ accesses data items $seats1$ and $seats2$ and these data items are passed to $Flight\_Res_{12}$ and $Flight\_Res_{12}$ uses these data items in its internal processing. Yet, because these tasks commit independently they are not executed within the scope of an isolation unit, i.e. a transaction, which provides isolation from other concurrently executing tasks. Other tasks of concurrently executing workflows can invalidate the data (e.g. $seats1, seats2$) being transferred between these tasks. Data-flow dependent tasks of a workflow such as $Check\_Cond_{11}$ and $Flight\_Res_{12}$ can be grouped into a single isolation unit.

In Example 2, a similar condition occurs at multiple sites. Because there is a serial control-flow and data-flow dependency between $Withdraw_{11}$ and $Deposit_{12}$ they must be executed in isolation and their serialization order must be compatible at every site that they have executed, that is, $Site_1$ and $Site_2$. So, either $Withdraw_{11}$ must be serialized after $Check_{211}$ at $Site_1$ or $Deposit_{12}$ must be serialized before $Check_{212}$ at $Site_2$. Note that *Report* tasks can be serialized in any order, since they do not affect the correct execution of other tasks. So, for example $Report_{13}$ should not necessarily have a consistent serialization order with $Wihdraw_{11}$ and $Deposit_{12}$ for the correctness.

To express these ideas precisely, a formal presentation of isolation units is given.

Definition 1. A **task** is a quadruple $t = (in, out, \eta, \varsigma)$ where in denotes the input parameters of task t; out denotes the output parameters of t; η is the name of t, and ς is the computation of the t. Actually computation ς is a tree on $O_{dm} \cup O_t$ where O_{dm} are the nodes representing the DM operations and O_t are the nodes corresponding to the abstract operations representing subtasks. □

Definition 2. There is a *data-flow dependency* between tasks t_i and t_j if $out_i \cap in_j \neq \emptyset$. The *data-flow dependency* is denoted as $t_i \models t_j$. □

In other words, at least one of the output parameters of t_i is mapped to an input parameter of t_j.

Definition 3. There is a *serial control-flow dependency*

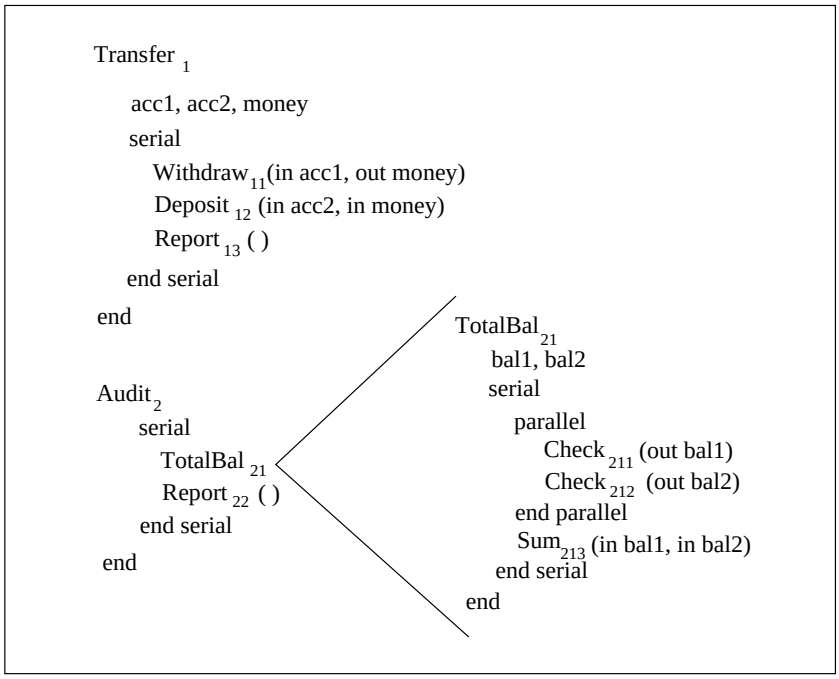

Fig. 3. Determining Isolation Units Using Data and Serial Control-Flow Dependencies

between tasks t_i and t_j if $t_j BCD t_i$. BCD [CR 91] denotes begin on commit dependency which means t_j can begin only after the commitment of t_i. The *serial control-flow dependency* is denoted as $t_i \Rightarrow_j$. □

Definition 4. Two tasks t_i and t_j belong to same *isolation unit* Π if $t_i \models t_j$ and $t_i \Rightarrow t_j$. □

The isolation units can be constructed automatically (i.e. without human intervention) by applying the Definition 4 repeatedly. Figure 3 represents an example to clarify Definition 4. In Figure 3, definitions of $Transfer_1$ and $Audit_2$ workflows of Example 2 are presented. $Transfer_1$ workflow contains three tasks to be executed in serial and $Audit_2$ contains two tasks namely $TotalBal_{21}$ and $Report_{22}$ to be executed in serial. Yet $TotalBal_{21}$ is a compound task that also includes two subtasks executing in parallel, namely $Check_{211}$ and $Check_{212}$. Starting with $Withdraw_{11}$, Π_1^1 of $Transfer_1$ contains $Withdraw_{11}$. Because $Withdraw_{11} \models Deposit_{12}$ and $Withdraw_{11} \Rightarrow Deposit_{12}$, Π_1^1 is augmented to $\{Withdraw_{11}, Deposit_{12}\}$. Finally, $\Pi_1^1 = \{Withdraw_{11}, Deposit_{12}\}$ and $\Pi_1^2 = \{Report_{13}\}$ since there is no *data-flow dependency* between $Report_{13}$ and other two. Similarly, since $Check_{211} \models Sum_{213}$, $Check_{211} \Rightarrow Sum_{213}$ and $Check_{212} \models Sum_{213}$, $Check_{212} \Rightarrow Sum_{213}$, $\Pi_2^1 = \{Check_{211}, Check_{212}, Sum_{213}\}$ and $\Pi_2^2 = \{Report_{22}\}$.

In the following section, isolation of nested tasks will be discussed.

4.2 Isolation of Nested Tasks

In our model, workflows may contain a *hierarchy of tasks*. In other words a compound task can contain any number of tasks and compound tasks. Therefore we need to deal with the isolation of hierarchically organized tasks. Nested tasks differ from flat tasks in that when two (sub)tasks are ordered this imposes an order between their parents. Thus isolation of tree of tasks must be defined. The theory provided in [HAD 97] for nested transactions in multidatabases is general enough to be applicable to workflow systems.

In the following, we will demonstrate how the ordering imposed by the leaf nodes are delegated to the upper nodes in the hierarchy. Note that, by assuming an imaginary root for all submitted workflows it is possible to model an *execution history of workflows*. *Execution history of workflows* is a tree on (sub)tasks and $\rightarrow$ is a nonreflexive and antisymmetric relation on the nodes of the tree. Actually, $\rightarrow$ is the ordering requirements on the leaf nodes due to execution order of conflicting DM operations. $\rightarrow$ satisfies the following axioms for any two (sub)tasks t_i and t_j:

i. transitivity: if $t_i \rightarrow t_j$ and $t_j \rightarrow t_k$ then $t_i \rightarrow t_k$
ii. delegation: if $t_i \rightarrow t_j$ and

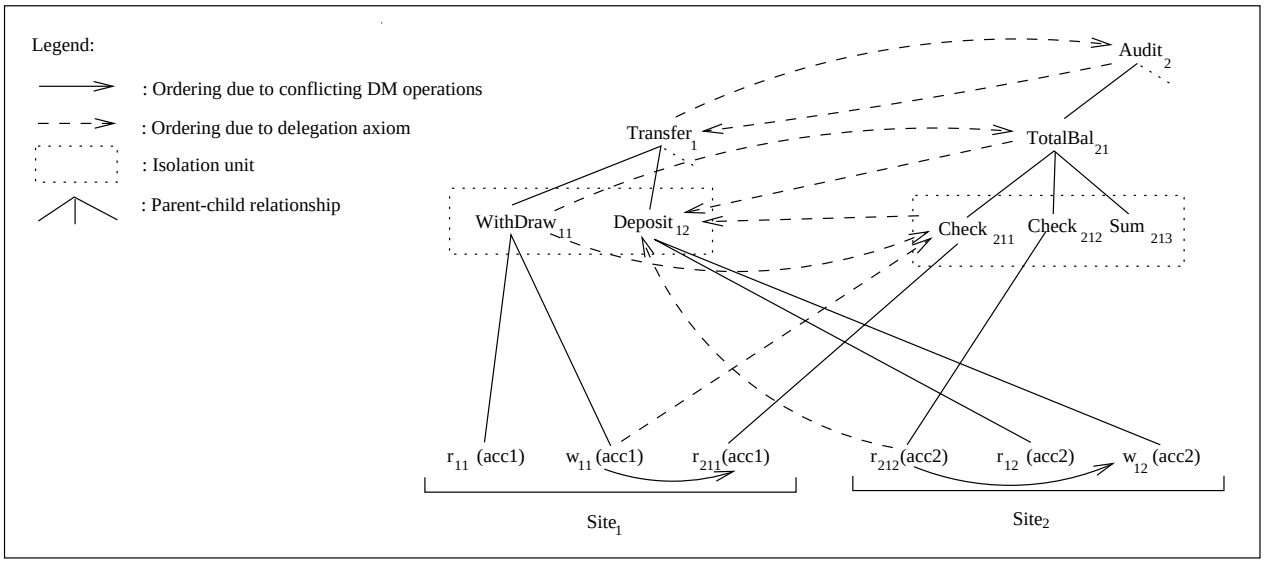

Fig. 4. Illustration of Delegation Axiom

a. if $parent(t_j) \notin ancestors(t_i)$ then $t_i \rightarrow parent(t_j)$

b. if $parent(t_i) \notin ancestors(t_j)$ then $parent(t_i) \rightarrow t_j$.□

Theorem 1. An *execution history of workflows* is serializable iff $\rightarrow$ is a partial order [3]. □

The proof of Theorem 1 is given in [HAD 97].

Consider the example in Figure 4. Isolation units are depicted within dotted rectangles in the figure. Since $Withdraw_{11}$ and $Check_{211}$ have issued conflicting DM operations on $acc1$ they are ordered as $Withdraw_{11} \rightarrow Check_{211}$ at $Site_1$ (The DM operations are not available at the workflow level; we obtain the related information from the data-flow by using the input, output parameters and from the serial control-flow dependencies). Also $Deposit_{12}$ and $Check_{212}$ are ordered as $Check_{212} \rightarrow Deposit_{12}$. Since, $Withdraw_{11}$ and $Check_{211}$ are ordered as $Withdraw_{11} \rightarrow Check_{211}$, $Withdraw_{11}$ and $TotalBal_{21}$ (which is $parent(Check_{211})$) are ordered as $Withdraw_{11} \rightarrow TotalBal_{21}$ (from Axiom i.a above). By applying the delegation definition repeatedly, the following order is obtained between $Transfer_1$ and $Audit_2$: $\{Transfer_1 \rightarrow Audit_2,\ Audit_2 \rightarrow Transfer_1\}$. $\rightarrow$ is not partial order here because its antisymmetry property is violated and the execution history for the isolation units in Figure 4 is not serializable. Some of the delegated orderings are not shown in Figure 4 for the sake of simplicity.

[3] Note that our partial order relation is irreflexive, antisymmetric and transitive

Now consider the case where tasks of Π_1^1 and Π_2^1 have consistent serialization orders at $Site_1$ and $Site_2$ as shown in Figure 5, i.e. $Withdraw_{11} \rightarrow Check_{211}$, $Deposit_{12} \rightarrow Check_{212}$. Hence, $Transfer_1 \rightarrow Audit_2$. Since $Report_{13}$ belongs to a different isolation unit (Π_1^2), its serialization order is independent from the tasks of Π_1^1 for correctness. Hence, the order due to $Report_{13}$ must be delegated to a different parent other than the parent of elements of Π_1^1, i.e. $Transfer_1$. In this way, $Report_{13}$'s inconsistent serialization order with $Withdraw_{11}$ and $Deposit_{12}$ does not effect $Withdraw_{11}$ and $Deposit_{12}$. Hence, parents of Π_1^1 and Π_1^2 are differentiated and a *virtual parent* for Π_1^2 is created and it is denoted as $Transfer_1'$. For the same reasons, $Audit_2'$ is created for Π_2^2. The point we want to make over here is the following: The correctness of an isolation unit can be checked and enforced by keeping its (sub)tasks under the same parent whereas the unrelated parts of the workflow can be executed freely by making them children of independent parents. So although the total execution of the workflow history in Figure 5 is not serializable, we make it semantically serializable by separating the parents of isolation units and delagating ordering relations due to different isolation units to different parents. Now, the order in Figure 5 is $\{Transfer_1 \rightarrow Audit_2,\ Audit_2' \rightarrow Transfer_1'\}$ which is serializable and correct from the application point of view.

As can be seen from the discussion presented above, the isolation units in a workflow can be identified and we claim that correctness measures can be applied on the basis of isolation units. This will allow for

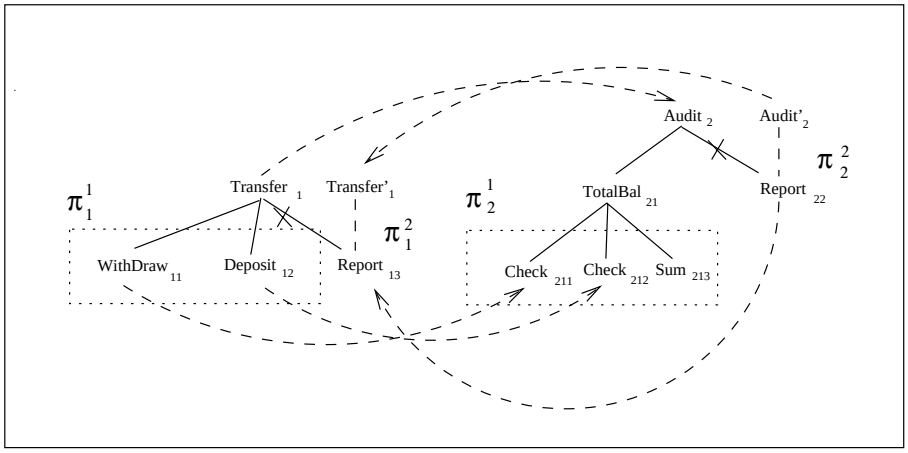

Fig. 5. Separating Parents of Different Isolation Units

the concurrent execution of rest of the workflow while preserving the correctness of isolation units.

In the following we will present a technique to provide for the correctness of concurrently executing nested tasks of workflow systems, based on isolation units.

5 Nested Tickets for Workflows

In this section, a technique for concurrency control of nested tasks of workflows, called *Nested Tickets (NT)* is presented. As described in [GRS 94], tickets determine the serialization orders of tasks. The main idea of NT technique is to give tickets to (sub)tasks at all levels, that is, both parent and child tasks obtain tickets. Then each (sub)task is forced into conflict with its siblings through its parent's ticket at all related sites. Note that since the parents of isolation units and unrelated parts of the workflow are separated only siblings within the same isolation unit are forced into conflict. The recursive nature of algorithm makes it possible to handle correctness of different task levels smoothly. The algorithm is fully distributed, in other words there is no central scheduler. This is due to each (sub)task knows its predetermined serialization order and behaves according to this order information.

To be able to provide a neat recursive algorithm, we imagine all the workflows to be children of a virtual task called OMNI. When OMNI task starts executing, it creates a siteTicket(OMNI) at each site whose default value is 0.

GlobalBegin(t_i) assigns a globally unique and monotonically increasing ticket number denoted as $TN(t_i)$ to all tasks denoted by t_i when they are initiated, that is, both the parent and the child tasks at all levels obtain a ticket. A Ticket Server provides tickets and guarantees that any new (sub)task obtains a ticket whose value is greater than any of the previously assigned ticket numbers. Since any child is submitted after its parent, this automatically provides that any child has a ticket number greater than its parent's ticket. When a (sub)task t_i starts at a local site, before it executes any of its operations, *LocalCheckTicket(t_i, k)* is executed at this site. Each child task reads the local ticket created by its parent at this site (this ticket is created for the children of $parent(t_i)$, i.e. $siblings(t_i)$), and checks if its own ticket value is greater than the stored ticket value in the ticket for $siblings(t_i)$ at this site. If it is not, the task t_i is aborted at all related sites and resubmitted. Otherwise, t_i sets the local ticket created by its parent to its own ticket value ($TN(t_i)$) and creates a site ticket, $siteTicket(t_i)$ with default value 0 for its children. As a result, all siblings of a (sub)task accessing to some $Site_k$ are forced into conflict through a ticket item created by the parent of these siblings at $Site_k$. This mechanism makes the execution order of all (sub)tasks of an isolation unit to be consistent at all related sites. In other words, the consistency of serialization order of the siblings of an isolation unit is provided by guaranteeing them to be serialized in the order of their ticket numbers. If a task is validated then its read and write operations on any item x are submitted to related RM.

The NT Algorithm:

GlobalBegin(t_i):
Get global ticket for t_i so that
$TN(t_i)$:=lastTicketNo+1;
lastTicketNo:=$TN(t_i)$; □

LocalCheckTicket(t_i, k):
If t_i is not OMNI then

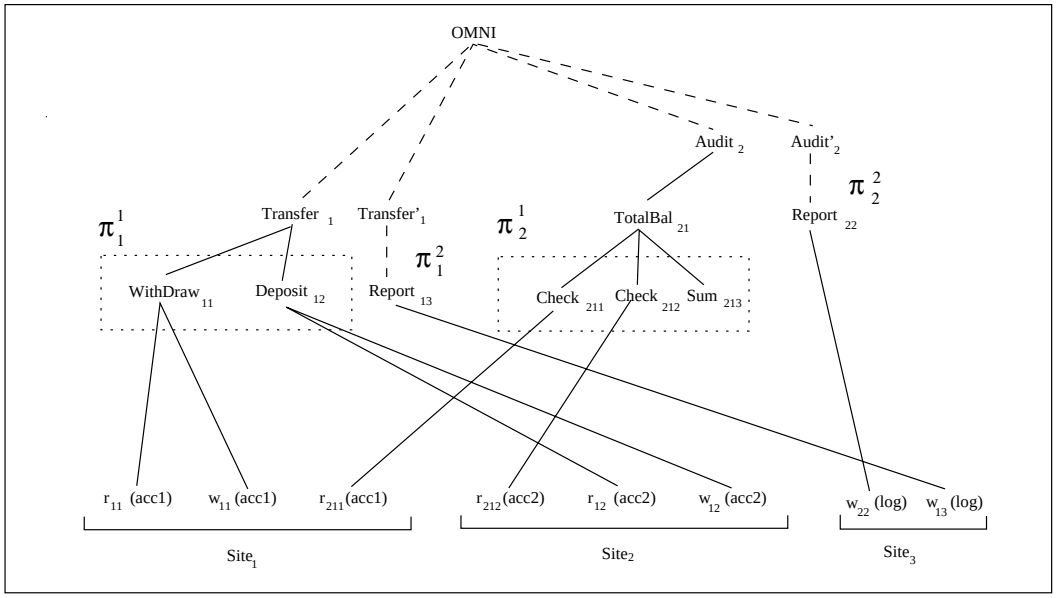

Fig. 6. Example of NT Technique

If siteTicket(parent(t_i)) > TN(t_i) then
Abort(t_i);
else
siteTicket(parent(t_i)):=TN(t_i);
Create(siteTicket(t_i)) at site k
with default value 0; □

In the following, an example is provided to clarify how NT technique is used to solve concurrency problems of workflow systems.

Example 3. Let us consider the example in Figure 6 and assume the tickets obtained from the Ticket Server to be as follows:

	TN		TN
OMNI	0	$Deposit_{12}$	7
$Transfer_1$	1	Sum_{213}	8
$Withdraw_{11}$	2	$Audit'_2$	9
$Audit_2$	3	$Report_{22}$	10
$TotalBal_{21}$	4	$Transfer'_1$	11
$Check_{211}$	5	$Report_{13}$	12
$Check_{212}$	6		

Execution at $Site_1$:
$Transfer_1$ is accepted since $siteTicket(parent(Transfer_1)) = siteTicket(OMNI) = 0 < TN(Transfer_1) = 1$ and $siteTicket(OMNI)$ is set to 1 and $siteTicket(Transfer_1)$ is created with default value 0. Since $siteTicket(parent(Withdraw_{11})) = 0 < TN(Withdraw_{11}) = 2$, $siteTicket(parent(Withdraw_{11}))$ is set to 2 and $r_{11}(acc1)$ and $w_{11}(acc1)$ are executed. Similarly $siteTicket(parent(Audit_2)) = siteTicket(OMNI) = 1 < TN(Audit_2) = 3$, $Audit_2$ is accepted and $siteTicket(OMNI)$ becomes 3 and $siteTicket(Audit_2)$ is created with default value 0. Next $TotalBal_{21}$ is accepted since $siteTicket(parent(TotalBal_{21})) = 0 < TN(TotalBal_{21}) = 4$ and $siteTicket(TotalBal_{21})$ is created with default value 0. $Check_{211}$ is also accepted and $r_{211}(acc1)$ is executed because $siteTicket(TotalBal_{21}) = 0 < TN(Check_{211}) = 5$.

Execution at $Site_2$:
$Audit_2$ is accepted since $siteTicket(parent(Audit_2)) = siteTicket(OMNI) = 0 < TN(Audit_2) = 3$ and $siteTicket(OMNI)$ is set to 3. $siteTicket(Audit_2)$ is created with 0 value. $TotalBal_{21}$ and $Check_{212}$ are accepted similarly and $r_{212}(acc2)$ is executed. Yet $Transfer_1$ at $Site_2$ is rejected and resubmitted to the system since $siteTicket(parent(Transfer_1)) = siteTicket(OMNI) = 3$ which is not less than 1.

Execution at $Site_3$:
$Audit'_2$ and $Report_{22}$ are accepted and $w_{22}(log)$ is executed. Now suppose that, $Transfer_1$ and $Audit_2$ are serialized consistently according to their ticket values at $Site_1$ and $Site_2$ and so $Transfer_1$ is accepted at $Site_1$. If parents of different isolation units are not differen-

tiated as in the original schedule, although tasks are executed correctly at all the sites, $Transfer_1$ would be rejected by the system. This due to $siteTicket\ (parent\ (Transfer_1)) = siteTicket\ (OMNI)$ is set 3 by $Audit_2$, hence it is not less than $TN\ (Transfer_1) = 1$ at $Site_3$. Since we differentiated parents of $Report_{13}$ and $Report_{22}$ the execution is as follows: $Audit'_2$ is accepted and $siteTicket\ (OMNI)$ is set to 9. $Report_{22}$ is accepted and $siteTicket\ (Audit'_2)$ is set to 10. Then $w_{22}(log)$ is executed. Similarly, $Transfer'_1$ is accepted since $siteTicket\ (parent\ (Transfer'_1)) = siteTicket\ (OMNI) = 9 < TN\ (Transfer'_1) = 11$. Finally, $Report_{13}$ is accepted and $w_{13}(log)$ is executed.

It can easily be shown that improper interleavings of the local transactions with the workflow tasks are also prevented with the NT technique. In fact, in [HAD 97] it is shown that NT Technique prevents improper interleaving of local transactions with global transactions.

6 Conclusions

To provide correctness in concurrently executing workflow systems, we have defined isolation units and provided a technique based on isolation units, for correctness of hierarchically structured workflows.

Formally, our model defines a task as a quadruple. Two (sub)tasks belong to same isolation unit (Π) if there is a data-flow and serial control-flow dependency between them. A (sub)task is said to execute correctly if it is ordered consistently with other (sub)tasks of its isolation unit at all related sites. To guarantee correct execution, each (sub)task at all levels is assigned a global ticket and it is expected that (sub)tasks are ordered according to their ticket values; otherwise they are aborted and resubmitted to the system.

Currently we are in the process of implementing this technique as a concurrency control service [A 97] for our Workflow Management System prototype, namely *MetuFlow*.

References

[A 97] B. Arpinar. Concurrency Control and Transaction Management in Workflow Management Systems. Ph. D. Thesis, in preparation, Dept. of Computer Engineering, Middle East Technical University, 1997.

[BBG 89] C. Beeri, P. A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transaction Systems. Journal of the ACM, 36(2), 1989.

[BDS 93] Y. Breitbart, A. Deacon, H. J. Schek, A. Sheth, and G. Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. ACM SIGMOD Record, 22(3), Sept. 1993.

[CR 91] P. K. Chrysanthis, and K. Ramamritham. A Formalism for Extended Transaction Models. In Proc. of the 17th Int. Conf. on VLDB, Barcelona, 1991.

[DHL 91] U. Dayal, M. Hsu, and R. Ladin. A Transaction Model for Long-Running Activities. In Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, September 1991.

[DS 93] D. R. McCarthy, and S. K. Sarin. Workflow and Transactions in InConcert. Special Issue on Workflow and Extended Transaction Systems, Bulletin of the Technical Committee on Data Engineering, Vol. 16, No. 2, June 1993.

[GHM 95] D. Georgakopoulos, M. Hornick, and F. Manola. Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation. IEEE Trans. on Knowledge and Data Eng., 1995.

[GHS 95] D. Georgakopoulos, M. Hornick, and A. P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases, 3, pp. 119-153, 1995.

[GRS 94] D. Georgakopoulos, M. Rusinkiewicz, and A. P. Sheth. Using Tickets to Enforce the Serializability of Multidatabase Transactions. IEEE Transactions on Knowledge and Data Engineering, 6(1), 1994.

[HAD 97] U. Halici, B. Arpinar, and A. Dogac. Serializability of Nested Transactions in Multidatabases. Intl. Conf. on Database Theory (ICDT '97), Greece, January 1997.

[L 83] N. A. Lynch. Multilevel Atomicity: A New Correctness for Database Concurrency Control. ACM Trans. on Database Systems, Vol. 8, No. 4, pp. 484-502, Dec. 1983.

[OMG 94] Object Transaction Service. OMG Document, 1994.

[RS 94] M. Rusinkiewicz, and A. P. Sheth. Transactional Workflow Management Systems. In Proceedings of Advances in Database and Information Systems, ADBIS'94, Moscow, May 1994.

[WR 92] H. Waechter, and A. Reuter. The ConTract Model. In Ahmed K. Elmagarmid, editor, Database Transaction Models for Advanced Applications, chapter 7, pp. 219-263, Morgan Kaufmann Publishers, San Mateo, 1992.

[W 91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. ACM TODS, 16(1), 1991.

[WWW 96] D. Wodtke, J. Weissenfels, G. Weikum, and A. K. Dittrich. The Mentor Project: Steps Towards Enterprise-Wide Workflow Management. In Proc. Twelfth Intl. Conf. on Data Eng., New Orleans, Louisiana, 1996.