Flexible Compensation of Workflow Processes

Weimin Du, Jim Davis, Ming-Chien Shan, Umesh Dayal
Software Technology Laboratory
HPL-96-72 (R.1)
February, 1997

E-mail: [du, davis, shan, dayal]@hpl.hp.com

workflow,
business process,
compensation

This paper addresses specific and implementation issues of workflow process compensation. The main consideration is to reduce the number of workflow activities that have to be compensated and re-executed when a failure occurs, as both can be very expensive. The main contributions of the paper are two-fold. First, we propose a flexible mechanism for specifying the compensation scope of a workflow process, based on a detailed analysis of dependencies between workflow activities. Second, we develop a novel implementation strategy that supports engine-based lazy compensation to further avoid unnecessary compensation efforts that are impossible to avoid at specification time. The proposed techniques are simple to implement, but also have the potential of significantly reducing compensation overhead.

Internal Accession Date Only

© Copyright Hewlett-Packard Company 1997

Flexible Compensation of Workflow Processes

Weimin Du, Jim Davis, Ming-Chien Shan, and Umesh Dayal
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
{du, davis, shan, dayal}@hpl.hp.com

Abstract

This paper addresses specification and implementation issues of workflow process compensation. The main consideration is to reduce the number of workflow activities that have to be compensated and re-executed when a failure occurs, as both can be very expensive. The main contributions of the paper are two-fold. First, we propose a flexible mechanism for specifying the compensation scope of a workflow process, based on a detailed analysis of dependencies between workflow activities. Second, we develop a novel implementation strategy that supports engine-based lazy compensation to further avoid unnecessary compensation efforts that are impossible to avoid at specification time. The proposed techniques are simple to implement, but also have the potential of significantly reducing compensation overhead.

1 Introduction

Workflow management systems (WFMSs) provide the ability to define and automate the flow of work through an organization to accomplish business tasks. Business tasks are first modeled (or re-engineered) as workflow processes, which are then automated by workflow management systems. A workflow process consists of coordinated executions of multiple steps (called *activities*) that may involve humans and require access to heterogeneous and distributed systems. Some applications also require selective use of transactional properties for individual tasks or entire workflow processes to meet consistency or reliability requirements of the business.

An important problem in managing workflow processes is that of failure recovery. The goal of recovery is to bring the failed process back to some semantically acceptable state so that the cause of the failure can be identified. The problem can then be fixed and the execution resumed with the hope that it will then complete successfully. Bringing the process back to a semantically acceptable state entails *compensating* – in some order – the already completed activities until the acceptable state is reached.

Compensation was first introduced in [GMS87] to simulate the transactional properties – atomicity, consistency, isolation, and durability – for long-running database applications that

would be too expensive to implement as single ACID transactions. The idea was to implement such an application as a *saga* or sequence of ACID transactions so that resources needed only at a particular stage could be released after the corresponding transaction completes. Atomicity was simulated by compensating already completed transactions in reverse order. Compensating a transaction t involves executing a transaction c that tries to undo the effects of t, essentially by restoring the data items written by the t to their states before t executed. Though simple, the concept of compensation has proven to be very effective in improving performance of long-running database applications.

Compensation of general workflow processes is more complex than for sagas. First, a workflow process is structurally more complex than a saga, and the execution of a process may establish quite complex control and data flow dependencies among the activities of the process. A workflow process specification may include conditional branching, concurrent execution of activities, loops and other complex control structures. As a result, identifying the compensation scope (i.e., which activities need to be compensated) for a general workflow process becomes a non-trivial task.

Second, a workflow process usually involves multiple independent database systems, application systems, and humans. The activities of a process typically perform arbitrarily complex operations, rather than simple database reads and writes. Thus, the activities can be very expensive to compensate and re-execute. It is therefore very important to minimize compensation scope to avoid unnecessary compensation effort.

1.1 Contributions of This Paper

This paper addresses both specification and implementation issues of workflow process compensation. The contributions of the paper are two-fold. First, we introduce a flexible mechanism for specifying the compensation scope of a workflow process, based on a detailed analysis of dependencies among the activities of the process. We propose different compensation scoping strategies (temporal-based, structural-based, computable, and extended) to reflect different business requirements and to meet different compensation needs. The goal of all these strategies is to minimize the amount of effort in compensating and re-executing the activities of a process. However, the applicability of the strategies depends on the semantics of the process. The process definer needs to indicate which strategy is acceptable, and the system can then automatically determines the best compensation scope. Second, we develop a novel implementation strategy, engine-based *lazy compensation*, which provides further optimization. This strategy allows the system to defer and perhaps eliminate unnecessary compensation and re-execution of activities, which may be impossible to detect at specification time.

The paper is organized as follows. We first present a workflow process model used in the paper and introduce our terminology. In Section 3, we introduce our basic compensation model and define the concept of acceptable compensations. In Section 4, we describe various strategies for determining compensation scope. Section 5 describes how to avoid compensation efforts discovered

at runtime to be unnecessary, using lazy compensation. Section 6 concludes the paper with a few remarks.

1.2 Related Work

A formal model of compensation for long-duration and nested transaction models has been described in [KLS90]. That paper assumes that the steps of the long duration transaction or nested transaction are themselves ACID database transactions that perform read and write operations against a database. We do not make such an assumption. The activities of our processes may or may not be transactions, and in general, they can perform arbitrarily complex operations against database systems or other systems.

A number of process models with compensation have been described in the literature [AAE+96, ELLR90, DHL91, LR94, EL95, BOH92, WR96]. All of these papers assume that the compensation scope is somehow predefined and that *all* the completed activities in the compensation scope are compensated. We argue in this paper that that is often wasteful, and that substantial savings can accrue by reducing the compensation scope. We show how to do this, based on a careful analysis of the dependencies among activities, and the semantics of the process. We show how lazy compensation can further reduce the amount of work to be done, by taking advantage of additional knowledge available at run time.

2 Workflow Process and Execution

As defined by Workflow Management Coalition [WfMC94], a workflow process is a coordinated set of workflow activities that are connected in order to achieve a common business goal. A workflow process is first specified using a workflow process model and then executed by Workflow Management Systems (WFMSs).

In this section, we first present the workflow process model used in the paper and also introduce our terminology. We then describe process execution. We assume here that a process executes successfully and thus no compensation needed. The next section will describe workflow process compensation to deal with failures.

2.1 Process Model

A workflow process is described as a directed graph $P = <\mathcal{N}, \mathcal{A}>$ comprising a set of nodes $\mathcal{N} = \{n_1, n_2, ...\}$, and a set of arcs $\mathcal{A} \subseteq \mathcal{N}^2$ connecting nodes in $\mathcal{N}$.

There are two kinds of nodes: *work nodes* and *control nodes*. Let $\mathcal{WN}$ and $\mathcal{CN}$ be the sets of work nodes and control nodes, respectively. Then $\mathcal{N} = \mathcal{WN} \cup \mathcal{CN}$.

A work node $wn \in \mathcal{WN}$ is a place holder for workflow activities. There are two kinds of

workflow activities: process and compensation activities. Each work node is associated with a process activity and an optional compensation activity.

A process activity is described as a logical representation of a piece of work contributing toward the accomplishment of a process. The specification of a process activity is mapped to the invocation of an operation on a business object during the execution of the containing work node. Each invocation may effect a manual operation by a human or a computerizable task to execute a legacy application, access databases, control instrumentation, sense events in the external world, or even effect physical changes.

A compensation activity is also a logical representation of a piece of work that semantically "undo" the effect of the corresponding process activity. Similar to process activities, a compensation activity is also mapped to the invocation of an operation on a business object which compensates the result of the business object operation corresponding to the process activity. For example, if the process activity is to charge a customer a certain amount of money, the compensation activity might be to give the customer a credit of the same amount.

A control node $cn \in \mathcal{CN}$ is used to define a process flow that is more complex than a simple sequence, such as concurrent process execution and synchronization of activities. A control node controls process flow according to the associated rules. Given a control node $cn \in \mathcal{CN}(P)$ and an outgoing arc $a = (cn, n) \in \mathcal{A}(P)$, where $n \in \mathcal{N}(P)$, we use $cn(a)$ to denote the set of rules of cn whose actions fire arc a.

As defined by the Workflow Management Coalition [WfMC94], there are three different kinds of data in a workflow process: *process-specific data* which are used only by the WFMS (e.g., for routing); *process-relevant data* which are used by both the WFMS and external applications; and *application-specific data* which are just used by external applications and are invisible to the WFMS. Given a process P, we use $\mathcal{SD}(P)$, $\mathcal{PD}(P)$ and $\mathcal{AD}(P)$ to denote the set of process-specific data, the set of process-relevant data, and the set of application-specific data set used in P, respectively. We also use $\mathcal{D}(P) = \mathcal{SD}(P) \cup \mathcal{PD}(P) \cup \mathcal{AD}(P)$ to denote the set of all data used in P.

Every node reads and writes data. A work node reads and writes only process-relevant data and application-specific data, while a control node reads and writes only process-relevant data and process-specific data. Given a node $n \in \mathcal{N}(P)$, we use $\mathcal{R}(n)$ and $\mathcal{W}(n)$ to denote the sets of data read and written by n, respectively. Given a control node $cn \in \mathcal{CN}(P)$ and an outgoing arc $a = (cn, n) \in \mathcal{A}(P)$, where $n \in \mathcal{N}(P)$, $\mathcal{R}(cn(a))$ is the set of data based on which cn decides if the arc a will be fired.

There are two kinds of arc: *forward arc* and *backward arc*. Let $\mathcal{FA}$ and $\mathcal{BA}$ be the sets of forward arcs and backward arcs, respectively, then $\mathcal{A} = \mathcal{FA} \cup \mathcal{BA}$.

A forward arc $a = (n_1, n_2) \in \mathcal{FA}$ represents the forward execution flow of workflow processes, where $n_1, n_2 \in \mathcal{N}$, while a backward arc is used to create loops in a workflow process. An important feature of the process model is that the forward arcs of a process form a directed

acyclic graph.

We say that a node $n_1 \in \mathcal{N}$ is reachable from another node $n_2 \in \mathcal{N}$ if they are connected by forward arcs and n_2 is the source of the path.

$$\texttt{Reachable}(n_1) = \{\ n \in \mathcal{N} \mid (n_1, n) \in \mathcal{FA}, \text{ or } \exists n_2 \in \mathcal{N} \text{ such that } n_2 \in \texttt{Reachable}(n_1) \text{ and } n \in \texttt{Reachable}(n_2)\ \}$$

Therefore, $\forall n_i \in \mathcal{N}, n_i \notin \texttt{Reachable}(n_i)$, and $\forall ba = (n_1, n_2) \in \mathcal{BA}$, where $n_1 \neq n_2$, $n_1 \in \texttt{Reachable}(n_2)$.

In summary, work nodes perform individual work, arcs define process flows, and control nodes control process flows.

2.2 Example Process Specification

Figure 1 shows an example process specification for SDH (Synchronous Digital Hierarchy) network configuration management [DSW95]. The process sets up network connections between two given end points in an SDH network. The process consists of seven work nodes (wn_{1-7}) and five control nodes (cn_{1-5}).

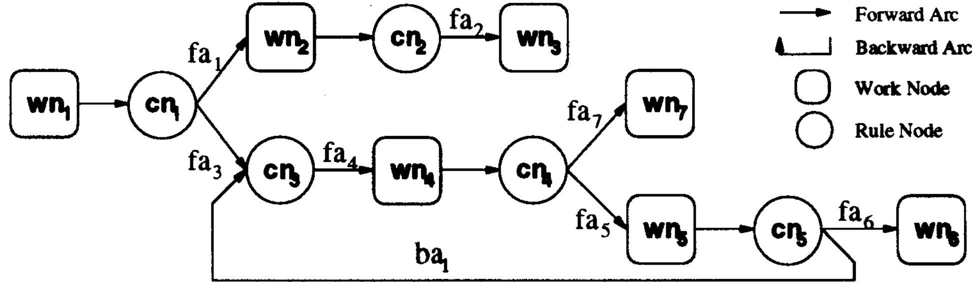

Figure 1: Example SDH Network Management Process Specification

wn_1 : Find two Add Drop Multiplexers (ADMs) close to the two end points.

cn_1 : Always fire both forward arcs fa_1 and fa_3, causing concurrent execution of wn_2 and cn_3.

wn_2 : Check for connections between the two end points and their corresponding ADMs.

cn_2 : Fire forward arc fa_2 if no connection exist between the end points and the end ADMs.

wn_3 : Add new lines to connect the end points to the end ADMs.

cn_3 : Always fire forward arc fa_4.

wn_4 : Find a route between the two end ADMs, check the capacity of the route and then confirm with the customer.

cn_3 : Fire forward arc fa_5 if there exists such a route. Also fire forward arc fa_7 if any ADMs in the route reached capacity threshold.

wn_5 : Configure the route by cross connecting ADMs in the route.

cn_5 : Fire forward arc fa_6 if the route has been configured successfully. Otherwise, fire backward arc ba_1 to find another route.

wn_6 : Create the trail by updating customer and configuration databases.

wn_7 : Order new equipments for future use.

Let P_0 be the above defined process. Then, $\mathcal{WN}(P_0) = \{wn_1, wn_2, wn_3, wn_4, wn_5, wn_6, wn_7\}$ and $\mathcal{CN}(P_0) = \{cn_1, cn_2, cn_3, cn_4, cn_5\}$. $ba_1 = (cn_5, cn_3)$ is the only backward arc and all others are forward arcs. $\texttt{Reachable}(cn_1) = \mathcal{N}(P_0) - \{wn_1\}$, while $\texttt{Reachable}(cn_4) = \{wn_5, cn_5, wn_6, wn_7\}$.

The example process P_0 will be used throughout the paper to illustrate concepts and terminology.

2.3 Process Ececution

Given a process specification P, an execution of P, denoted $\mathcal{P} = \{i_1, i_2, ...\}$, is a set of control node instances and activity instances, where for each $i \in \mathcal{P}$, there either exists an $cn_k \in \mathcal{CN}(P)$ such that i is the execution of cn_k, or exists an $wn_k \in \mathcal{WN}(P)$ such that i is the execution of the process activity of wn_k. In general, we use i_k to denote the instance of (control or work) node $n_k \in \mathcal{N}(P)$. We also use e_k to denote the instance of control node $cn_k \in \mathcal{CN}(P)$ and use t_k and c_k to denote the instances of process activity and compensation activity, respectively, of work node $wn_k \in \mathcal{WN}(P)$.

For the sake of simplicity, we assume that activities of a workflow process are executed sequentially. The order in which activities are executed is termed the execution order. Every node instance $i \in \mathcal{P}$ has a start and a complete times. Start and complete times of a node instance i are denoted as $\texttt{Start}(i)$ and $\texttt{End}(i)$, respectively. Clearly, $\texttt{Start}(i) < \texttt{End}(i)$ and $\texttt{End}(i_1) < \texttt{Start}(i_2)$ if i_1 precedes i_2 in the execution order. The assumption will be relaxed later in the paper.

Given a process specification, there can be many different executions. We only consider process executions that are consistent with the specification. More specifically, $\forall n_1, n_2 \in \mathcal{N}(P)$, $\texttt{End}(i_1) < \texttt{Start}(i_2)$ if $i_2 \in \texttt{Reachable}(i_1)$. In Figure 1, for example, node wn_3 is always executed after cn_2. In the paper, we express a process execution as a sequence of process activity and control node instances, in the execution order. Often, we omit control node instances from process execution for brevity. For example, the following is an execution of process P_0.

$$E_1 : t_1, t_2, t_3, t_4, t_5, t_6$$

The following definition defines acceptable process executions[1]. We assume that all process

[1]Strictly speaking, the definition handles only processes without backward arcs. However, it can be easily extended to include backward arcs by considering different executions of an activity in a loop as different activity instances. Please also note that the definition is very strict and allows only AND synchronization. That is, a node

activities complete successfully. The definition will be extended in the next section to deal with failed process activity instances.

Definition 2.1 (Acceptable Process Executions With no Compensation)
A process execution $\mathcal{P}$ is an acceptable process execution of a process specification P if

- $\forall i \in \mathcal{P}, \exists n \in \mathcal{N}(P)$ *such that i is an execution of n;*
- $\forall (n_1, n_2) \in \mathcal{FA}(P)$, $\texttt{Start}(i_2) > \texttt{End}(i_1)$,
- *Given $n_2 \in \mathcal{N}(P)$, $\forall n_1 \in \mathcal{N}(P)$ such that $(n_1, n_2) \in \mathcal{FA}(P)$ and $i_1 \in \mathcal{P}$, then $i_2 \in \mathcal{P}$.*

Since $wn_3 \notin \texttt{Reachable}(wn_4)$ and $wn_4 \notin \texttt{Reachable}(wn_3)$, the WFMS can execute wn_3 and wn_4 in any order. Therefore, E_1 is an acceptable process executions of P_0, and so is E_2.

$$E_2 : t_1, t_4, t_2, t_5, t_3, t_6$$

On the other hand, the following execution is not acceptable as $wn_3 = \texttt{Reachable}(wn_2)$, but t_3 precedes t_2 in the execution order:

$$E_3 : t_1, t_4, t_3, t_2, t_5, t_6, t_7$$

3 Workflow Process Compensation

Compensation is a technique to deal with process activity failures[2]. When a process activity instance failed, the WFMS is responsible for bringing the process execution to a designated *end compensation point.* An end compensation point is a previous execution step of the process which represents an acceptable intermediate execution state and hopefully also a decision point where certain actions can be taken to either fix the problem that caused the failure or choose an alternative execution path to avoid the problem[3]. Note that compensation and later re-execution of a work node is considered semantically equivalent to the original execution, but may not be identical.

The objective of this paper is to devise correct and efficient compensation strategies for workflow processes. A simple approach is to compensate all previously completed instances that

will not be executed until the nodes from which it is reachable have all completed. The definition can be easily extended to also allow OR synchronization but this extension is omitted for brevity.

[2]Note that compensation is different from backward arcs which provide alternative executions. The latter represents controlled process behavior, while the former represents abnormal (or failed) process execution. In process P_0, for example, wn_5 may complete sucessfully but returning a negative result (i.e., the route cannot be configured) which is handled by backward arc ba_1. Work node wn_5 may also fail, due to, e.g., situations beyond the capability of cross connection program. When this happens, wn_5 will fail and has to be compensated.

[3]See [DDS96] for specifications of end compensation points.

happened after the end compensation point and in the exact reverse order of the original execution. This is the approach generally adopted by database applications (see, e.g., Sagas [GMS87]). The approach, however, may not be efficient, as it could also compensate instances that do not really need compensation.

Consider, for example, process execution E_2 in Section 2, Suppose that the execution failed at wn_5 and the end compensation point is wn_4. Since wn_2 was scheduled between wn_4 and wn_5, the above simple approach will compensate (and re-execute) not only wn_4 and wn_5, but also wn_2. Activity wn_2 which checks the two end connection points, represents a task that is independent of wn_4 and wn_5, which search for a route between the two end connection points. The compensation and re-execution of wn_4 and wn_5 may result in a different route between the two end connection points. The result of wn_2, however, will always remain the same and therefore it does not need compensation.

While the simple compensation approach is generally acceptable for database applications, it can be very costly for workflow applications. This is because database operations are simple and can be efficiently compensated and re-executed, but workflow activities are generally complicated and also expensive to compensate. A better compensation approach for workflow processes is therefore to compensate only the instances that really need compensation (e.g., wn_4 and wn_5 in E_2).

Efficient compensation, on the other hand, is not easy, due to the complicated structure and semantics of workflow processes. It requires not only detailed analysis of the process specification, but also knowledge of process semantics. An important observation is that correct and exact workflow process compensation is not achievable by the WFMS alone. The main reason is that process execution, especially that of process activities, is not totally under WFMS' control. It is therefore essential for process designers to provide certain knowledge about workflow processes. For example, the information that a work node (i.e., activity instances) will never be affected by another work node via application specific data is very useful for the WFMS to determine if the former should be compensated when the latter has been compensated and re-executed. The information is otherwise invisible to the WFMS, as it has no access to application specific data. The issue for WFMSs is to allow process designers to provide the information (hopefully minimal) in an easy way and effectively make use of the information.

A fundamental issue in efficient workflow process compensation is to ensure correctness. Informally, a compensation is acceptable if and only if all affected activity instances have been compensated, and the compensation (of all instances) represents a semantic inverse of the original execution. The purpose of this section is to establish correctness criteria for completeness and semantic equivalence in the traditional sense. The following two sections will present specific techniques that relax the criteria to allow more efficient compensation than the traditional simple strategy.

3.1 Complete Compensation

Since the end compensation point is always compensated and re-executed (to fix the problem that caused the failure), all the instances that have been affected should also be compensated. Thus, a compensation strategy is complete if all previously completed activity instances directly or indirectly affected by the end compensation point are compensated.

3.1.1 Node Dependencies

In general, a node can be affected by another node via either *data dependency* or *execution dependency*. A node is directly affected by another node via data dependency if the former reads data updated by the latter. Data dependency is important as re-execution of a node may produce different data values.

Definition 3.1 (Data Dependency)
Given a process execution $\mathcal{P}$ of a process P and $n_1, n_2 \in \mathcal{N}(P)$, i_1 is affected by i_2 via data dependency, denoted $i_2 \xrightarrow{dd} i_1$, if

- $\texttt{End}(i_2) < \texttt{Start}(i_1)$, *and*
- $\exists d \in \mathcal{D}(P) \cap \mathcal{W}(n_2) \cap \mathcal{R}(n_1)$.

We use $\xrightarrow{dds}$ to denote the transitive closures of data dependency.

A node is directly affected by another node via execution dependency if the execution of the latter implies that of the former. Execution dependency is important as nodes have to be compensated if they will not be re-executed, even if they are not affected (via data dependency) by the end compensation point.

Definition 3.2 (Execution Dependency)
Given a process execution $\mathcal{P}$ of a process P and $wn_1, wn_2 \in \mathcal{WN}(P)$, t_1 depends on t_2 via an execution dependency, denoted $t_2 \xrightarrow{ed} t_1$, if $\exists cn_3 \in \mathcal{CN}(P)$ such that

- $\texttt{Start}(e_3) > \texttt{End}(t_2)$,
- $a = (cn_3, wn_1) \in \mathcal{FA}(P)$,
- $\mathcal{W}(wn_2) \cap \mathcal{R}(cn_3(a)) \neq \emptyset$, *and*
- $\exists d \in \mathcal{W}(wn_2) \cap \mathcal{R}(cn_3(a))$ *such that* $\forall n_4 \in \mathcal{N}(P)$, *where* $\texttt{End}(t_2) < \texttt{Start}(i_4)$ *and* $\texttt{End}(i_4) < \texttt{Start}(e_3)$, $d \notin \mathcal{W}(n_4)$.

In other words, t_1 is affected by t_2 via execution dependency if t_2 affects the rule node that fires the inward arc of t_1. The dependency is important, as the execution status of t_1 may be affected by the compensation and re-execution of t_2. For example, t_1 was executed in the original execution, as the result of t_2 execution. But e_3 may choose a different execution path that does not include t_1 at re-execution time, as t_2 may produce different results that affect e_3's decision. Therefore, t_1 may have to be compensated (but not re-executed) as the result of t_2's compensation and re-execution.

We use $\xrightarrow{eds}$ to denote the transitive closure of direct execution dependency.

We also say that i_1 directly depends on (or is directly affected by) i_2, denoted $i_2 \rightarrow i_1$, if $i_2 \xrightarrow{ed} i_1$ or $i_2 \xrightarrow{dd} i_1$. We use $\xrightarrow{*}$ to denote the transitive closures of $\rightarrow$.

3.1.2 Compensation Complete Executions

Given a process specification P and a work node $wn_k \in \mathcal{WN}(P)$, we use c_k to denote the compensation activity instance of wn_k. Using the dependencies defined above, a compensation complete execution can be defined as follows.

Definition 3.3 (Compensation Complete Executions)
A process execution $\mathcal{P}$ of P is compensation complete if $\forall wn_1, wn_2 \in \mathcal{WN}(P)$, such that $t_2 \xrightarrow{*} t_1$ and $c_2 \in \mathcal{P}$, then $c_1 \in \mathcal{P}$.

Consider the following process execution (with compensation) for the process given in Figure 1.

$$E_4 : t_1, t_4, t_2, t_3, t_5, t_6, c_6, c_4, t_4, t_6.$$

$wn_4 \xrightarrow{ed} wn_5$ and $t_4, c_4, t_5 \in \mathcal{P}_0$, but $c_5 \notin \mathcal{P}_0$. According to the definition, E_4 is not compensation complete.

E_5 and E_6 below, on the other hand, are both compensation complete.

$$E_5 : t_1, t_2, t_3, t_4, t_5, t_6, c_6, c_5, c_4, t_4, t_5, t_6.$$
$$E_6 : t_1, t_2, t_3, t_4, t_5, t_6, c_5, c_6, c_4, t_4, t_5, t_6.$$

3.2 Order Preserving Compensation

Another important aspect of correct process compensation is compensation order. Basically, the compensation sequence (e.g., c_5, c_6, c_4 in E_6) should represents a semantic inverse of the original execution (e.g., t_4, t_5, t_6 in E_6).

In general, a particular sequence of process activity execution is a reflection of process semantics (via data or execution dependencies). Consider, for example, process P_0 in Figure 1. wn_4 must precede wn_5 because a route may only be configured after it has been found. wn_5

must precede wn_6 because updating customer database before actually configuring the route may present to customers an inconsistent execution state. To preserve the process semantics, process activities should be compensated in the reverse order of the original execution. Compensating wn_5 before wn_6, for example, may result in the same inconsistent execution state as executing wn_6 before wn_5.

To define order preserving compensation, let us first introduce the notion of compensation equivalence transformation. The basic compensation equivalence has the following two rules.

Rule 1. $t_k, t_j, c_j \Leftrightarrow t_k$, where $k \neq j$.

Rule 2. $t_k, c_k, t_k \Leftrightarrow t_k$.

The first rule says that executing two process activities one after another and then compensating the latter one is semantically equivalent to just executing the former. The second rule says that executing a process activity, compensating it, and then re-executing it is semantically equivalent to just executing the activity once. They both state the same basic assumption of compensation: a compensation activity is a semantic inverse of the corresponding process activity.

The general strict compensation equivalence is defined as the transitive closure of $\Leftrightarrow$ (denoted as $\overset{*}{\Leftrightarrow}$). Therefore, a process execution $\mathcal{P}$ is strict compensation equivalent to another execution $\mathcal{P}'$ (denoted as $\mathcal{P} \overset{*}{\Leftrightarrow} \mathcal{P}'$) of the same process specification if $\mathcal{P}$ can be transformed to $\mathcal{P}'$ via a sequence of basic (strict compensation equivalence) transformations.

For example, E_5 can be transformed to E_1 via the following basic compensation transformations.

$$
\begin{array}{rl}
t_1, t_2, t_3, t_4, \underline{t_5, t_6, c_6}, c_5, c_4, t_4, t_5, t_6 & \text{(Rule 1: } t_5, t_6, c_6 \Rightarrow t_5) \\
t_1, t_2, t_3, \underline{t_4, t_5, c_5}, c_4, t_4, t_5, t_6 & \text{(Rule 1: } t_4, t_5, c_5 \Rightarrow t_4) \\
t_1, t_2, \underline{t_3, t_4, c_4}, t_4, t_5, t_6 & \text{(Rule 1: } t_3, t_4, c_4 \Rightarrow t_3) \\
t_1, t_2, t_3, t_4, t_5, t_6 &
\end{array}
$$

E_6, however, cannot be transformed to E_1, or any other process execution with no compensation activities.

Note that both basic transformations require that the compensation activity instance immediately follow the corresponding process activity instance. This essentially guarantees that process activities are compensated in the exact reverse order of the original execution. Process executions with this property are termed order preserving compensations.

Definition 3.4 (Order Preserving Compensation)
Given a process specification P and a process execution $\mathcal{P}'$ (with compensation) of P. $\mathcal{P}'$ is order preserving if there exists an acceptable process execution $\mathcal{P}$ (with no compensation) of P such that $\mathcal{P}' \overset{*}{\Leftrightarrow} \mathcal{P}$.

By the definition, E_5 is order preserving, as $E_5 \overset{*}{\Leftrightarrow} E_1$ and E_1 is an acceptable execution.

3.3 Acceptable Process Compensation

Completeness and order preserving are two aspects of correct process compensation. As we have seen, there are executions that are complete but not order preserving. There are also executions that are order preserving but not complete. These two properties together guarantee the correctness of process execution and compensation.

Definition 3.5 (Acceptable Process Execution With Compensation)
A process execution $\mathcal{P}$ is an acceptable process execution if it is both compensation complete and order preserving.

According to the definition, E_6 is not an acceptable process execution of P_0 as it is not order preserving. E_5, on the other hand, is an acceptable process execution of P_0.

As we mentioned, Definition 3.5 only defines acceptable process execution in the traditional sense. We could therefore prove the simple compensation strategy mentioned in Section 3. It is however too strict in that it excludes many semantically acceptable executions. For example, the following execution is not acceptable according to the definition, however, as we argued in Section 3, it should be considered as a semantically acceptable execution.

$$E_7 : t_1, t_4, t_2, t_3, t_5, t_6, c_6, c_5, c_4, t_4, t_5, t_6.$$

The next two sections will relax the definition to include executions like E_7, by making use of process semantics.

4 Compensation Scoping Strategy

Given a process execution $\mathcal{P}$ of a process specification P. A compensation scope is a subset of $\mathcal{P}$ that includes activity instances to be compensated. Compensation scoping strategies compute compensation scopes for given process executions and failed activity instances. Since an activity instance needs to be compensated only if it is affected (directly or indirectly) by the end compensation point, the compensation scope can be expressed as a function of the end compensation point.

This section presents five scoping strategies, based on a more detailed analysis of data and execution dependencies. The purpose is to provide more flexible ways of computing compensation scopes whenever permitted by process semantics. The bottom line is therefore to ensure completeness of compensation. We describe scoping strategies (as functions of end compensation points) and discuss conditions under which they ensure completeness.

4.1 Minimal Compensation Scope

The minimal compensation scope depends upon the execution history including application-specific and process-relevant data as well as the exact state of the business process. Given an end compensation point $wn_2 \in \mathcal{WN}(P)$, the minimal compensation scope for t_2, denoted $\mathcal{MCS}(t_2)$, is the subset of $\mathcal{P}$ whose members depend on t_2 via both data and execution dependencies.

Definition 4.1 $\mathcal{MCS}(t_2) = \{i \in \mathcal{P} \mid t_2 \xrightarrow{*} i\}$.

Theorem 4.1 *The minimal scoping strategy is compensation complete.*

Proof: Since $t_2 \xrightarrow{*} t_2$, $t_2 \in \mathcal{MCS}(t_2)$. Assume that $c_2 \in \mathcal{P}$ and, $\forall wn_1 \in \mathcal{WN}(P)$, $t_2 \xrightarrow{*} t_1$. By Definition 4.1, $t_1 \in \mathcal{MCS}(t_2)$. Therefore, $c_1 \in \mathcal{P}$ and by Definition 3.3, the strategy is compensation complete. □

Minimal compensation scope is also the smallest scope that results in correct compensation. In other words, it does not include any compensation that is always unnecessary. This is the case as it includes only nodes that are affected by re-execution of the end compensation point via data and execution dependency. Unfortunately, minimal compensation scope is neither computable by the WFMS at process specification time nor at runtime, as application-specific data is invisible to the WFMS and therefore it is impossible for the WFMS to figure out the application-specific data dependencies.

4.2 Temporal-Based Compensation Scope

Temporal-based compensation scope includes all nodes that are started after the end compensation point had completed. The temporal-based compensation scope for $t_2 \in \mathcal{P}$, where $wn_2 \in \mathcal{WN}(P)$, denoted $\mathcal{TCS}(t_2)$, is a subset of $\mathcal{P}$ whose members started after t_2 had completed.

Definition 4.2 $\mathcal{TCS}(t_2) = \{i \in \mathcal{P} \mid \texttt{End}(t_2) < \texttt{Start}(i)\}$.

Theorem 4.2 *The temporal-based scoping strategy is compensation complete.*

Temporal-based scoping is guaranteed to be complete because all nodes that started execution after the end compensation point are compensated. Temporal-based scoping is the only strategy that is both computable and guaranteed to be safe, independent of the structure and previous execution of the process. It is also the easiest to compute because the scope can be determined by simply searching the system log file. The disadvantage, of course, is the heavy compensation overhead due to unnecessary compensation.

4.3 Computable Compensation Scope

Computable compensation scope is a subset of minimal compensation scope that is defined using only dependency information available to the WFMS. Since data dependencies are in general not computable by the WFMS, we need to further classifying data dependencies according to their visibility to the WFMS. As we mentioned, both process specific and process relevant data are visible to the WFMS, while application specific data are not.

Definition 4.3 (Visible Data Dependency)
Given a process execution $\mathcal{P}$ of a process P and $n_1, n_2 \in \mathcal{N}(P)$, i_1 directly depends on i_2 via a visible data dependency, denoted $i_2 \xrightarrow{dd} i_1$, if

- $\texttt{End}(i_2) < \texttt{Start}(i_1)$, *and*
- $\exists d \in (\mathcal{SD}(P) \cup \mathcal{PD}(P)) \cap \mathcal{W}(n_2) \cap \mathcal{R}(n_1)$ *such that* $\forall n_3 \in \mathcal{N}(P)$, *where* $\texttt{End}(i_2) < \texttt{Start}(i_3)$ *and* $\texttt{Start}(i_1) > \texttt{End}(i_3)$, $d \notin \mathcal{W}(n_3)$.

We use $\xrightarrow{vd}$ to denote all dependencies visible to the WFMS and $\xrightarrow{vd*}$ to denote its transitive closure. Then $i_2 \xrightarrow{vd} i_1$ if either $i_2 \xrightarrow{ed} i_1$ or $i_2 \xrightarrow{vdd} i_1$.

The computable compensation scope for $t_2 \in \mathcal{P}$, where $wn_2 \in \mathcal{WN}(P)$, denoted $\mathcal{CCS}(t_2)$, is a subset of $\mathcal{P}$ whose members depends on t_2 via visible data and execution dependency.

Definition 4.4 $\mathcal{CCS}(t_2) = \{i \in \mathcal{P} \mid t_2 \xrightarrow{vd*} i\}$.

The computable strategy is interesting in that it provides the best estimation of the minimal compensation scope based on information the WFMS has. In other words, computable compensation scope is the minimal scope (i.e., $\mathcal{CCS}(t_2) = \mathcal{MCS}(t_2)$) if there is no application specific data dependency between nodes. The disadvantage is that the compensation scope is difficult to compute (even at runtime), as execution dependencies are determined by rule node semantics.

Theorem 4.3 *The computable scoping strategy is compensation complete if* $\forall t_1, t_2 \in \mathcal{P}, t_1 \xrightarrow{dds} t_2$ *implies* $t_1 \xrightarrow{vdds} t_2$.

Proof: Since $t_2 \xrightarrow{*} t_2$, $t_2 \in \mathcal{CCS}(t_2)$. Assume that $c_2 \in \mathcal{P}$ and $t_2 \to t_1$. Then $t_2 \xrightarrow{ed} t_1$ or $t_2 \xrightarrow{vdd} t_1$. Thus, $t_2 \xrightarrow{vd} t_1$. Similarly, $t_2 \xrightarrow{*} t_1$ implies $t_2 \xrightarrow{vd*} t_1$. By Definition 4.4, $t_1 \in \mathcal{CCS}(t_2)$. Therefore, $c_1 \in \mathcal{P}$ and by Definition 3.3, the strategy is compensation complete. □

4.4 Structural-Based Compensation Scope

Structural-based strategy defines compensation scope based on the structure of the process specification. The following structure dependency provides a simulation of execution dependency that can be easily computed at specification time. The assumption is that in most cases, a structure dependency does imply either a data or an execution dependency, as otherwise the structure dependency could be removed to improve parallelism.

Definition 4.5 (Structure Dependency)
Given a process execution $\mathcal{P}$ of a process P and $n_1, n_2 \in \mathcal{N}(P)$, i_1 directly depends on i_2 via structure dependency, denoted $i_2 \xrightarrow{sd} i_1$, if $n_1 \in \texttt{Reachable}(n_2)$.

Similarly, we use $\xrightarrow{sd*}$ to denote the transitive closures of $\xrightarrow{sd}$

The structural-based compensation scope for $t_2 \in \mathcal{P}$, where $wn_2 \in \mathcal{WN}(P)$, denoted $\mathcal{SCS}(t_2)$, is the subset of $\mathcal{P}$ whose members depends on t_2 via structure dependency.

Definition 4.6 $\mathcal{SCS}(t_2) = \{i \in \mathcal{P} \mid t_2 \xrightarrow{sd*} i\}$.

Theorem 4.4 *The structure-based scoping strategy is compensation complete if $\forall i_1, i_2 \in \mathcal{P}, i_1 \not\xrightarrow{dd*} i_2$.*

Proof: Since $t_2 \xrightarrow{*} t_2$, $t_2 \in \mathcal{SCS}(t_2)$. Assume that $c_2 \in \mathcal{P}$ and, $\forall wn_1 \in \mathcal{WN}(P)$, $t_2 \xrightarrow{*} t_1$. Since $\xrightarrow{dd*}$ is empty, $t_2 \xrightarrow{ed*} t_1$. Thus, $t_2 \xrightarrow{sd*} t_1$. By Definition 4.6, $t_1 \in \mathcal{SCS}(t_2)$. Therefore, $c_1 \in \mathcal{P}$ and by Definition 3.3, the strategy is compensation complete. □

This strategy is the most easy to compute and can be completed at process specification time. The assumption is that structurally independent nodes usually do not affect each other. Please note that the assumption may not be true for many processes. It should be used only when the process designer is sure that no other nodes will be affected that do not structurally depend on the end compensation point.

The computation of $\mathcal{SCS}(t_2)$ is done in two steps. First at specification time, the WFMS identifies all nodes that are reachable from wn_2, forming a superset of $\mathcal{SCS}(t_2)$. At run time, the WFMS checks the system log file to remove from the set those nodes that have not yet been started (i.e., not recored in the log), resulting in $\mathcal{SCS}(t_2)$.

4.5 Extended Compensation Scope

The extended strategy defines a bigger compensation scope than does the structure-based strategy by including nodes affected via process-relevant and process-specific data.

We say that i_1 directly depends on (or is directly affected by) i_2 via extended dependency, denoted $i_2 \xrightarrow{xd} i_1$, if $i_2 \xrightarrow{sd} i_1$ or $i_2 \xrightarrow{vdd} i_1$. We use $\xrightarrow{xd*}$ to denote the transitive closure of $\xrightarrow{xd}$.

Definition 4.7 $\mathcal{XCS}(t_2) = \{i \in \mathcal{P} \mid t_2 \xrightarrow{xd*} i\}$.

The extended strategy is similar to computable strategy, but is easier to compute, as it does not use semantic information of rule nodes. Similar to computable strategy, the extended strategy guarantees correct compensation only if nodes do not depend on each other via application specific data.

Theorem 4.5 *The extended scoping strategy is compensation complete if* $\forall t_1, t_2 \in \mathcal{P}, t_1 \xrightarrow{dd*} t_2$ *implies* $t_1 \xrightarrow{vdd*} t_2$.

4.6 Summary

In summary, minimal compensation scope is the best but is generally impossible for the WFMS to compute. The temporal-based scope is the only strategy that is both computable and is guaranteed to be complete under all conditions. It should therefore be the WFMS' default choice when no strategy has been specified. The WFMS may also support structure-based, extended and computable scoping strategies as an optimization to temporal-based compensation scope. It is the process designers' responsibility to properly use these strategies. Improper uses of structure-based, extended, or computable strategies may result in incorrect compensation.

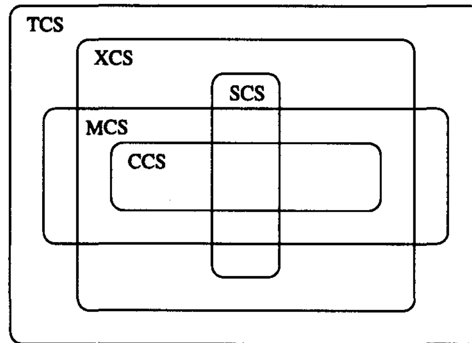

Figure 2: Comparison of Compensation Scoping Strategies

The relationship among the five scoping strategies is given in the following theorem (Figure 2).

Theorem 4.6 *Given a process executions* $\mathcal{P}$ *of a process specification* P *and* $i \in \mathcal{P}$, *then*

- $\mathcal{CCS}(i) \subseteq \mathcal{MCS}(i) \subseteq \mathcal{TCS}(i)$.
- $\mathcal{SCS}(i) \subseteq \mathcal{XCS}(i) \subseteq \mathcal{TCS}(i)$.
- $\mathcal{CCS}(i) \subseteq \mathcal{XCS}(i) \subseteq \mathcal{TCS}(i)$.

Consider the following compensation scenario for process P_0 defined in Figure 1. Suppose that wn_6 failed in execution E_2 $(t_1, t_2, t_4, t_5, t_3, t_6)$ and wn_5 is the end compensation point. In other words, t_6 failed to create the trail as configured in t_5) and the process will be rolled-back to t_5 to try different cross connections. Suppose that t_3 (connecting the end points to the end ADMs) and t_5 (cross connecting the two end ADMs) conflict with each other as they share the same data (i.e., the ports of the end ADMs at which the end points are connected and the end ADMs are cross connected).

There are two different ways of handling the situation.

1. Compensating both t_5, t_6 (i.e., removing the cross connection) and t_3 (i.e., disconnecting end points from the end ADMs) and re-executing t_5, t_3, t_6:

$$t_1, t_2, t_4, t_5, t_3, t_6, c_6, c_3, c_5, t_5, t_3, t_6.$$

2. Compensating t_5 and t_6 only and reusing the previous ADM ports:

$$t_1, t_4, t_2, t_5, t_3, t_6, c_6, c_5, t_5, t_6.$$

It is not hard to see that both executions are (semantically) acceptable. The second alternative is more efficient but the re-execution of wn_5 may fail, due to possibly unavailable cross connection between the two ADM ports. When this happens, the process execution will have to be rolled-back to t_4 which forces the compensation and execution of t_3. The first alternative, on the other hand, is more expensive but safer (i.e., use only the ADM ports that can be cross connected). The two alternatives can be easily specified by choosing temporal-based or structural-based scoping strategies for wn_6:

$$\mathcal{TCS}(wn_5) = \{wn_3, wn_5, wn_6\}.$$
$$\mathcal{SCS}(wn_5) = \{wn_5, wn_6\}.$$

5 Lazy Compensation

Section 4 of the paper has discussed various scoping strategies that allow the WFMS to compute compensation scopes at specification time or runtime. The presented compensation scoping strategies allow us to avoid, under certain conditions, the compensation and re-execution of many nodes that can be statically proved not being affected by the end compensation point. On the other hand, not all nodes that statically depend on the end compensation point (via, e.g., data or structure dependencies) necessarily imply compensation.

We now describe another technique that we call *lazy compensation* to address the problem. Lazy compensation assumes a compensation scope chosen by one of the strategies described in section 4, but can further reduce compensation effort. Lazy compensation makes use of run time

execution state including input data to each process activity as well as additional declarations by the workflow process designer. The basic idea of lazy compensation is to defer compensation of process activities whenever possible until re-execution time. The assumption is that most process activities that are compensated will eventually be re-executed and many such re-executions are semantically identical to the original execution. The hope is that compensation and re-execution of some nodes can be saved altogether if the compensation can be deferred and the re-execution is in some sense semantically the same as the original execution.

Let us use the SDH network configuration management process P_0 to illustrate the idea. Consider work nodes wn_3, wn_4 and wn_5. Work node wn_3 depends on wn_4 as it has to use the ports (of the end ADMs) that can be cross connected to the ports (of the same end ADMs) chosen by wn_4. Work node wn_5 also depends on wn_4 as it must configure the route selected by wn_4. Suppose that wn_3 follows wn_4 but precedes wn_5 in the execution and wn_5 fails. In other words, the process has found a route between the two end ADMs, established connections between the end points to the selected ports at the end ADMs, but the application that configures the route failed. Thus, process execution should be rolled back to wn_4 to fix the problem and be re-executed. Since there are many different routes between two given ports (of the two end ADMs), it is very likely that wn_4 will find a different route between the same ports. Thus, wn_4 and wn_5 need compensation and re-execution. Work node wn_3, on the other hand, may or may not need compensation, depending on run time execution state (i.e., the ADM ports selected by wn_4).

Unfortunately, wn_3 depends on wn_4 via data dependency (i.e., the ADM ports). According to the compensation scoping strategies and acceptable process executions defined in Sections 2 and 3, it must always be compensated. On the other hand, if we can defer the compensation of wn_3 until re-execution time, then we will be able to tell if wn_3 needs compensation by looking at the input data (i.e., the ADM ports selected by wn_4). The following executions show lazy compensation of wn_3 where d_3 represents deferring compensation of wn_3.

$$E_8 : t_1, t_2, t_4, t_3, t_5, c_5, d_3, c_4, t_4, t_3, t_5, t_6.$$
$$E_9 : t_1, t_2, t_4, t_3, t_5, c_5, d_3, c_4, t_4, c_3, t_3, t_5, t_6.$$

In E_8, the re-execution of wn_4 selects the **same** ADM ports. This means that wn_3 need not be compensated, as it will produce semantically equivalent data (or even exact the same output data if wn_4 is deterministic). In E_9, however, the re-execution of wn_4 **has** selectsed defferent ADM ports. This means that wn_3 **must** be compensated and re-executed. Note thet E_8 can result in substantially less work being performed by the process.

Lazy compensation clearly violates the correctness criteria we described in Section 2.3, as process activities are not compensated in the reverse order of original execution. For example, E_8 is not compensation complete and E_9 is not order preserving. On the other hand, they both represent semantically acceptable process executions.

The purpose of this section is to relax the previously defined correctness criteria to include

acceptable lazy compensations. First we describe conditions under which lazy compensation can effectively and correctly avoid unnecessary compensation effort. Next we present algorithms for lazy compensation, and also give an informal correctness proof using extended compensation equivalence rules.

5.1 Lazy Compensation Group

There are two reasons why acceptable compensation should be complete and order preserving. First, a process activity may affect or be affected by other process activities of the same workflow process, according to the order in which they are executed. Second, a compensation activity may be nondeterministic. It may produce different results at different executions, even if the input data are the same. The only way that always ensures acceptable process executions is to compensate in the exact reverse order of the original execution.

There are however other ways to ensure correct process executions, especially if the process designer can make strong statements about the process and compensation activities. For example, if a work node is *isolated*, in that it does not affect and also is not affected by other activities, and its compensation activity is *deterministic* and represents a *true inverse* of the corresponding process activity, then its compensation can be deferred until re-execution time. At re-execution time, the WFMS will decide if compensation and re-execution is really necessary, by comparing the original and current execution states.

For a given work node, we distinguish between two execution states: pre-image which is the portion of the process relevant data provided to the node, and post-image which is the portion of the process relevant data generated by the node. If the pre-image at the re-execution time is the same as that at the original execution time, compensation is not necessary. This is the case for the following two reasons. First, whether the work node is compensated has no effect on other nodes, as it is isolated. Second, since the compensation activity is deterministic and a true inverse of the corresponding process activity, and the two pre-images are the same, compensation and re-execution guarantee to produce a semantically acceptable post-image.

For real workflow applications, many activities do interact with each other. However, they often interact in a clustered way. More specifically, a cluster of work nodes interact but only with each other. For example, wn_4, wn_5, and wn_6 interact with each other by sharing the same route configuration, stored in an application database. However, other activities do not need to access route configuration data. This has motivated us to introduce the notion of *lazy compensation group (LCG)*. An LCG is a set of work nodes in a process specification that have the the following properties:

1. All activities in the LCG have no interactions with activities outside the LCG except through input/output data (i.e., process relevant data that are visible to the WFMS); and
2. All compensation activities in the LCG guarantee that their original input state will be

restored upon compensation. In other words, they are deterministic and true inverses of the corresponding process activities with respect to the input/output data. Note that they may store additional non-inverted data in an outside application database accessible only by members of this LCG.

Suppose that work nodes wn_4, wn_5 and wn_6 of process P_0 form an LCG. The only outside interaction is with wn_3 but via process relevant data (i.e., selected ports of the end ADMs). The compensation activity of wn_5, for example, is to disconnects the ADM cross connections corresponding to the route. This compensation could clearly be implemented as a deterministic, true inverse of the process activity (cross connection). Similarly, wn_3 forms an LCG of its own. Therefore, wn_3 can be compensated more flexibly. More specifically, c_3 can be deferred until re-execution time without affecting other work nodes. Therefore, we can argue that E_9 is acceptable according to the above generalization.

A few things worth mentioning about lazy compensation groups.

First, process activities are free to apply non-deterministic behavior. It is only compensation activities that are constrained to act as a true inverse with regards to the process relevant data. Not all activities need fall into an LCG. An LCG might also have only one member.

Second, the undeferral and compensation of activities in an LCG must be done in the reverse order of the completion of their original executions. This is true because although these activities don't interact outside of their group, they are allowed to interact with each other (e.g., share the same application database). Note that activities not in an LCG are **not** constrained to act as true inverses to their corresponding process activities. Thus the compensation of a process activity t_i, not in any LCG may force the delayed compensation of all of the deferred activities in every LCG containing a member that is to be compensated after t_i.

Third, there are usually several different ways to form LCGs in a process. In general, the smaller the cardinality of each LCG, the greater the opportunity for compensation optimization. LCGs therefore should be chosen with cardinality as small as possible and to partition the subset of nodes that do participate in LCGs, to eliminate as much unnecessary compensation as possible.

5.2 Lazy Compensation Algorithm

We now describe our lazy compensation algorithms. The algorithms consist of two major parts: one for node compensation (`LazyCompensate`) and the other for re-execution (`ExecuteNode`).

For each work node wn_x to be compensated, `LazyCompensate` does one of the following three things:

1. *Immediate compensation*, if wn_x is not in any LCG.
2. *Defering compensation*, if wn_x is in an LCG and the execution state (i.e., CurrentPostImage) is the same as that of the original execution (i.e., OriginalPostImage). The hope is that

compensation is not needed if the pre-image also remains the same in the re-execution. Since compensation activity is a true inverse, it always maps the same post-image to the same pre-image.

3. *Group compensation*, if the post-images are different. Different post-images imply possibly different pre-images. Compensation, therefore cannot be deferred, as re-execution is necessary (to produce the new post-image). At this point, not only the current work node, but also all deferred work nodes in the same LCG will have to be compensated, as they all interact with the current work node.

For each LCG we will maintain an ordered list $LCG_i.deferredList$ of the compensations deferred within that group.

```
LazyCompensate( wn_x ){
    compute CurrentPostImage
    lookup OriginalPostImage
    lookup OriginalPreImage
    if ( ∀i, wn_x ∉ LCG_i ) { /* case 1 */
        NormalCompensate( wn_x); /* Whatever is done for non-Lazy compensation*/
        return;
    }
    LCG_i = LCG in which wn_x resides
    if ( CurrentPostImage = OriginalPostImage ) { /* case 2 * /
        add wn_x to end of LCG_i.deferredList list of deferrals
        update process data from OriginalPreImage
    } else { /* case 3 */
        for each element wn_y in LCG_i.deferredList in order {
            NormalCompensate( wn_y );
            pop wn_y from LCG_i.deferredList
        }
        NormalCompensate( wn_x );
    }
}/* End of LazyCompensate() */
```

Execution of a work node wn_x needs to be modified in two ways in order to support lazy compensation. When a node to be executed belongs to a LCG any of whose compensations are still queued for deferral then we need to take special actions.

If the node to be run is at the end of the deferred queue (meaning it was the least recently deferred compensation in that LCG not undeferred or rendered *useful*) and was originally run with the same pre-image then we may pop that queue item and render this item *useful*. If the node to be run not the end of the deferred queue or if the pre-images do not match, then we must

undefer and compensate all items remaining in the queue prior to running the node in the normal way.

At this time, the current pre-image will be compared with the original one. Compensation (and thus re-execution) will be omitted if the two are the same. The work node will be removed from the deferredList. Otherwise, all remaining work nodes of the LCG will be compensated (in the exact reverse order of the original execution).

```
ExecuteNode( wn_x ){
    compute CurrentPreImage
    if ( ∃i such that wn_x ∈ LCG_i ) {
        while ( LCG_i.deferredList ≠ ∅){
            wn_y = last item on LCG_i.deferredList
            if (( wn_y.nodeId = wn_x.nodeId )
            and ( CurrentPreImage = wn_y.OriginalPreImage )) { /* case 1 */
                pop wn_y from LCG_i.deferredList
                update process data from wn_y.OriginalPostImage
                return;
            } else  { /* case 2 */
                for each element wn_y in LCG_i.deferredList in order {
                    NormalCompensate( wn_y );
                    pop wn_y from LCG_i.deferredList
                }
            }
        }
    }
    dispatch the actual activity      ...
    obtain the returned PostImage
    update process data from the PostImage
}/* End of ExecuteNode() */

TerminateProcessInstance( process p ){
    for each LCG_i{
        while ( LCG_i.deferredList≠ ∅){
            pop wn_y from LCG_i.deferredList
            NormalCompensate( wn_y );
        }
    }
}/* End of TerminateProcessInstance() */
```

Note that not all deferred compensations result in savings. Deferred compensations may be canceled for the following four reasons.

1. During the compensation we may discover that the compensation of some activity not in any LCG does not restore the output parameters for some other activity t_x in an LCG. At that point we must undefer and actually compensate all deferred compensations in that LCG (case 3 of `LazyCompensate`).

2. During later re-execution, we may come to re-execute an activity t_x in an LCG and find that its inputs no longer match those inputs provided during the initial execution whose compensation was deferred d_x. At that point we must undefer and actually compensate all remaining deferred compensations in that LCG (case 2 of `ExecuteNode` when CurrentPreImage $\neq wn_y$.OriginalPreImage).

3. During later re-execution, we may come to execute an activity t_x in an LCG and find this activity does not have a deferred compensation, yet there are deferred compensations in that LCG. At that point we must undefer and actually compensate all remaining deferred compensations in that LCG (case 2 of `ExecuteNode` when wn_y.nodeId $\neq wn_x$.nodeId).

4. As the process finishes, we may find that some deferred compensations have not yet been re-executed. These deferred compensations must be undeferred and actually compensated (`TerminateProcessInstance`).

Please also note that a work node whose compensation is deferred may not be re-executed at all as it may fall on a branch of the process that is not taken during the re-execution. In such cases, the deferral must be canceled and the activity compensated when it is discovered that it will not be re-executed. This time can be just prior to the process termination or when any activity in the same LCG runs a non-deferred forward execution.

5.3 Lazy Compensation Examples

Consider again the example process P_0 described in section 2.2 and shown in Figure 1. As we mentioned, wn_4, wn_5, and wn_6 form an LCG and wn_3 alone forms an LCG. Suppose that, after an initial process execution of E_1 $(t_1, t_2, t_3, t_4, t_5, t_6)$, we have a failure at t_6 which requires rollback to wn_4. Consider the following cases.

If the compensation c_4 and re-execution t_4 of wn_4 selected the same route (e.g., wn_4 fixed problem that caused wn_6 to fail), compensation and re-execution of wn_5 and wn_6 can be safely omitted, as the same cross connection can be re-used, saving two compensations and two re-executions.

$$E_{10} : t_1, t_2, t_3, t_4, t_5, t_6, d_6, d_5, c_4, t_4, t_5, t_6.$$

Note that we still mark t_5 in the E_{10} when the system re-executes wn_5, even though that re-execution is not performed other than to remove d_5.

If c_4 and t_4 did not select the original route, wn_5 and wn_6 have to be actually compensated.

$$E_{11} : t_1, t_2, t_3, t_4, t_5, t_6, d_6, d_5, c_4, t_4, c_6, c_5, t_5, t_6.$$

This case shows optimistic deferred compensations that may turn out to be required.

If c_4 and t_4 selected the same route but with different price, then only the compensation of wn_5 can be omitted, as it only depends on wn_4. Work node wn_6, on the other hand, has to be compensated as it needs the new price to update customer database.

$$E_{12} : t_1, t_2, t_3, t_4, t_5, t_6, d_6, d_5, c_4, t_4, t_5, c_6, t_6.$$

Now let us consider the following initial execution which failed at wn_5.

$$E_{13} : t_1, t_2, t_4, t_3, t_5.$$

If c_4 and t_4 selected a different route but used the same end ADM ports, then wn_5 has to be compensated, but not wn_3.

$$E_{14} : t_1, t_2, t_4, t_3, t_5, d_5, d_3, c_4, t_4, c_3, t_3, t_5.$$

The case shows that the more LCGs provided by the process designer, the more the savings. Since wn_3 is in its own LCG, it need not be compensated even though wn_5 did need to be compensated and re-executed.

5.4 Lazy Compensation Equivalence

Let us call the process executions generated by lazy compensation algorithms (Section 5.2) *process executions with lazy compensation* (PELCs). As we have mentioned, PELCs (e.g., E_{10} in Section 5.3) are semantically acceptable but are not compensation complete and order preserving. We now revisit the compensation equivalence by taking into account lazy compensation. We first extend previously defined basic equivalence rules, followed by an informal correctness proof of the lazy compensation algorithms using the extended compensation equivalence rules.

Let E be a PELC. First, we notice that E can be considered as semantically complete in that there always exists a compensation complete process execution E' such that E and E' are semantically equivalent. Recall that compensation is omitted only for work nodes of LCGs and when re-execution state (i.e., pre- and post-images) is the same as that of the original execution. The isolation and deterministic properties of deferred compensation activities imply that compensation and then re-execution of the work nodes are semantically equivalent to null operations.

To argue that work nodes in LCGs can also be deferred, as we did in lazy compensation algorithms, we need the following new compensation equivalence rules, where α is a segment of process execution.

Rule 3. $d_x, \alpha, t_x \Leftrightarrow c_x, \alpha, t_x$, where $c_x, t_x \notin \alpha$ and $wn_x \in LCG_i$.

Rule 4. $d_x, \alpha, c_x \Leftrightarrow c_x, \alpha$, where $c_x, t_x \notin \alpha$.

Rule 5. $c_k, c_l \Leftrightarrow c_l, c_k$, where $wn_l \in LCG_i$ and $wn_k \notin LCG_i$.

Rule 6. $t_k, c_l \Leftrightarrow c_l, t_k$, where $wn_l \in LCG_i$ and $wn_k \notin LCG_i$.

The third rule says that deferring and then re-execution of a work node with the same execution state is equivalent to compensating and then re-execution of the same work node if it belongs to an LCG. In other words, it is acceptable to delay compensation of a work node if it belongs to an LCG. The fourth rule says that it is acceptable to first defer and then undefer and actually compensate a work node. We argue that the two rules are valid due to special properties of LCGs as previously explained. The last two rules express that a compensation in an LCG commutes with activities not in that same LCG. Again this is valid as work nodes in an LCG are isolated from others outside the LCG.

The extended compensation equivalence is defined as the transtive closure of all six basic and extended compensation equivalence rules.

Due to space limitation, correctness of lazy compensation algorithms is only informally outlined below using examples.

First, please note that (1) the algorithm never inserts actual compensations out of order within an LCG (case 3 of `LazyCompensate`); and (2) every compensation inserted onto the deferred list is eventually either re-executed or undeferred (`TerminateProcessInstance`).

Consider a PELC and a deferred work node wn_x of the PELC. Since the algorithms always insert either c_x or t_x for d_x, d_x can either be reduced by Rule 3 to a c_x, or removed by Rule 4. Each of those rules leaves a c_x in the process execution. The remaining process execution consists only of process and compensation activities. In addition, it consists of exactly the same process and compensation activities as the non-Lazy compensation process execution, although in a different order. Rule 5 allows us to commute each c_x into its correct position (except for work nodes in the same LCG), producing a process execution equivalent to the non-lazy compensation with the same scope.

The following transformation sequence shows how E_{14} is transformed to an acceptable process execution:

$$t_1, t_2, t_3, t_4, t_5, t_6, \underline{d_6, d_5, c_4, t_4, t_5, c_6}, t_6 \quad \text{(Rule 4: } d_6, d_5, c_4, t_4, t_5, c_6 \Rightarrow c_6, d_5, c_4, t_4, t_5)$$
$$t_1, t_2, t_3, t_4, t_5, t_6, c_6, \underline{d_5, c_4, t_4, t_5}, t_6 \quad \text{(Rule 3: } d_5, c_4, t_4, t_5 \Rightarrow c_5, c_4, t_4, t_5)$$
$$t_1, t_2, t_3, t_4, t_5, t_6, c_6, c_5, c_4, t_4, t_5, t_6$$

The following execution cannot be generated by the algorithms, as wn_5 and wn_6 are not compensated in the right order. The execution cannot be transformed to any acceptable process execution by extended compensation equivalence rules, as c_5 and c_6 do not commute.

$$t_1, t_2, t_3, t_4, t_5, t_6, d_5, \underline{d_6, c_4, t_4, t_5, c_6}, t_6 \quad \text{(Rule 4: } d_6, c_4, t_4, t_5, c_6 \Rightarrow c_6, c_4, t_4, t_5)$$
$$t_1, t_2, t_3, t_4, t_5, t_6, \underline{d_5, c_6, c_4, t_4, t_5}, t_6 \quad \text{(Rule 3: } d_5, c_6, c_4, t_4, t_5 \Rightarrow c_5, c_6, c_4, t_4, t_5)$$
$$t_1, t_2, t_3, t_4, t_5, t_6, \underline{c_5, c_6}, c_4, t_4, t_5, t_6 \quad \text{(!!!)}$$

6 Conclusion

The paper has addressed specification and implementation issues of workflow process compensation. The main consideration is to reduce the number of workflow activities that have to be compensated and re-executed, as both can be very expensive. The problem is difficult due to: (1) the complexity of workflow process structures, and (2) the impossibility for a workflow management system to compute the minimal compensation scope.

Our approach is for the workflow management system to provide flexible compensation scoping strategies. These strategies rely on process designers to properly use them. Unnecessary compensation can be avoided at specification time by choosing a proper scoping strategy, and at runtime by detecting unnecessary compensation and re-execution (engine-based lazy compensation). The proposed techniques are simple to implement and easy to use. Nevertheless, they have the potential to significantly reduce compensation overhead.

The proposed work is conducted in the context of HP OpenPM, a research prototype developed at HP Labs [DDS95]. The proposed techniques have been partially (and will be completely) implemented in OpenPM. We believe that the techniques are general enough to be used by similar workflow management systems. They could also be used in some advanced database applications where long running transactions are used.

In Section 2, we have assumed sequential execution of workflow activities. The assumption can be relaxed to allow concurrent execution in several ways. One approach is to implement each activity as an ACID (Atomic, Consistent, Isolated, and Durable) transaction. The WFMS coordinates activity instances so that they can be serialized (as in database transaction executions). A serialized process execution can be logically considered as a sequential one, in the serialization order.

For more general cases where activities are not ACID transactions, the assumption can be relaxed as follows. If an activity instance is executed entirely before another (e.g., $\mathbf{End}(t_1) < \mathtt{Start}(t_2)$), they should be compensated in the exact reverse order (e.g., $\mathbf{End}(c_2) < \mathtt{Start}(c_1)$). Otherwise, they can be compensated in any order, or even concurrently. The fact that they are concurrently executed implies that the execution order is not important. Thus, compensation order is also unimportant.

Despite its importance, workflow process compensation has received little attention so far. The results presented in the paper are preliminary and many issues still remain open. Due to space limitations, many important issues were not covered, or only discussed briefly. For example, we have focused on compensation of process activities (work nodes). An important issue is the compensation of events (both internal and external). Another interesting issue is specification of end compensation points. As we mentioned, end compensation points are special work nodes that are responsible for detecting and fixing (or avoiding) problems that caused failures. An activity may fail for many different reasons. Fixing end compensation points may result in either unnecessary compensation (too conservative) or incorrect compensation (too optimistic).

Dynamic adjustment of end compensation, which we call compensation cascading [DDS96], is thus necessary. Further research, however, is needed on these and other issues.

References

[AAE+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan. *Advanced Transaction Models in Workflow Contexts*, Data Engineering 1996.

[AARS87] P. Attie, M. Aingh, M. Rusinkiewicz and A. Sheth. *Specification and Enforcing Intertask Dependencies*, VLDB 1993.

[BK85] B. Blaustein and C. Kaufman. *Update Replicated Data During Communication Failures*, VLDB 1985.

[BOH92] A. Buchmann, M. Ozsu, M. Hornick, D. Georgakopoulos and F. Manola. *A Transaction Model for Active Distributed Object Systems*, A. Elmagarmid (ed) Transaction Model for Advanced Database Applications, Morgan-Kaufmann, 1992

[CR91] P. Chrysanthis and K. Ramamritham. *A Formalism for Extended Transaction Models*, VLDB 1991.

[DDS95] J. Davis, W. Du and M. Shan. *OpenPM: An Enterprise Process Management System*, IEEE Data Engineering Bulletin, 1995.

[DDS96] J. Davis, W. Du and M. Shan. *Flexible Compensation of OpenPM Workflow Processes*, Technical Report, HPL-96-72, 1996.

[DHL91] U. Dayal, M. Hsu and R. Ladim. *A Transactional Model for Long Running Activities*, VLDB 1991.

[DSW95] W. Du, M. Shan and Chris Whitney. *SDH Network Management with OpenPM*, Data Engineering 1996.

[EL95] J. Eder and W. Liebhart. *The Workflow Activity Model WAMO*, Coopis, 1995.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. *A Multidatabase Transactional Model for Interbase*, VLDB 1990.

[Gray81] J. Gray. *The Transaction Concept: Virtues and Limitations*, VLDB 1981.

[GMS87] H. Garcia-Molina and K. Salem. *Sagas*, SIGMOD 1987.

[HR87] T. Haerder and K. Rothermel. *Concepts for Transaction Recovery in Nested Transactions*, SIGMOD 1987.

[KLS90] H. Korth, E. Levy and A. Silberschatz. *A Formal Approach to Recovery by Compensation Transactions*, VLDB 1990.

[KS94] N. Krishnakumar and A. Sheth. *Specification of Workflow with Heterogeneous Tasks in Meteor*, VLDB 1994.

[KYH95] K. Karlapalem, H. Yeung and C. Hung. *CapBasED-AMS: A Framework for Capability-based and Event-driven Activity Management System*, Coopis 1995.

[Leym95] F. Leymann. *Frank Leymann: Supporting Business Transactions Via Partial Backward Recovery In Workflow Management Systems*, BTW, 1995.

[LR94] F. Leymann and D. Roller. *Business Process Management with Flowmark*, COMPCON 1994.

[MS93] D. McCarthy and S. Sarin. *Workflow and Transactions in InConcert*, Data Engineering Bulletin, 1993.

[WR96] H. Wachter and A. Reuter. *The ConTract Model*, A. Elmagarmid (ed) Transaction Model for Advanced Database Applications, Morgan-Kaufmann, 1992

[WfMC94] Workflow Management Coalition. *Glossary: A Workflow Management Coalition Specification*, Workflow Management Coalition Standard, 1994.