ISSN 1816-6075 (Print), 1818-0523 (Online)
Journal of System and Management Sciences
Vol. 1 (2011) No. 5, pp. 49-65

Document recovery in a business process using activity dependency

Tianming Gan[1], Hyerim Bae[2]
[1] Logistic Information Technology Dpt., Pusan National University, South Korea
tianming.gan@pusan.ac.kr
[2]Industrial Engineering Dpt., Pusan National University, South Korea
hrbae@pusan.ac.kr

Abstract: This paper focuses on the problem of bringing a business process document to a consistent state after exceptional occurrences. Although many traditional recovery methods can support exception handling, they are mainly concerned with process backtracking, forward stepping, and compensation. However, prior to the use of recovery algorithms, it is essential to find a rollback activity for organizations that aim to reduce their costs and increase their revenues. In this paper, two aspects of this issue were addressed: an algorithm that considers the inverse relationship between reparability and reactivity in finding a rollback activity, and a document recovery mechanism that recovers a document to a consistent version. A prototype system was implemented. Simulation results generated in an analysis of the various decision weight parameters showed that our mechanism can support different rollback activity decisions and document recovery strategies.

Keywords: Exception Handling, Rollback Activity, Activity Dependency, Document Recovery, Business Process Management

1. Introduction

The Business Process Management System (BPMS) is widely used to automate business processes characterized by increasingly variegated information technology (Bae and Kim, 2007). Nowadays, BPMS is an essential technology for organizations in the conduct of their daily operations. However, the system's capability needs to be extended to deal with resources, especially data in the form of documents which used in the execution of business processes. Documents have been considered to play important roles in business processes,

as they are widely used as data carriers by many organizations (Bae and Kim, 2002).

Consistent and reliable execution of business processes is crucial for all organizations. Nevertheless, due to increasingly complex, dynamic and error-prone operating environments, it is extremely challenging for either enterprise managers or process designers to determine all of the possible combinations of exceptions or to design corresponding handling methods (Wang and Sun, 2010, chap. 11). Therefore, a flexible, systematic and autonomic approach for exception handling is essential for the success of complex BPMS applications in wider fields.

In order to deal adequately with exceptions, one of the most important things is to find a rollback point. When we find a rollback activity, there is a trade-off between consistency and time. That is, although a farther rollback makes for a greater possibility of fixing problems, more time will be required to cancel and redo activities. The motivation of the present work was the necessity of finding a rollback activity enabling development of a flexible and dynamic BPMS to support the growing number of exceptions that cannot be designed in advance. There were three major goals:

Develop an algorithm for finding a rollback activity from which the recovery system can achieve a trade-off between the two decision variables (i.e., reparability and reactivity);

Propose a document recovery mechanism that can enable a process to maintain document consistency at all times, even after a failure;

Implement a prototype system and demonstrate how it can support different strategies under mutable situations.

The rest of this paper is organized as follows: Section 2 discusses the exception handling issues and briefly summarizes the previous work. Section 3 presents preliminaries to be used for further study. The algorithms used for finding rollback and recovery activities are introduced in Sections 4 and 5, respectively. Section 6 discusses a document recovery mechanism. Section 7 treats the prototype system implementation and experimentation. Finally, Section 8 summarizes the conclusions and future work.

2. Background

2.1 Problem Description

A process of holiday travel planning is illustrated in Figure 1. The principal can choose the travel dates, destination (California or Hongkong), and amusement.

There are several latter possibilities for each destination, such as Disneyland or sailing in Hongkong. Then, the principal books a hotel, flight, and tickets for amusements. Additionally, for convenience, the hotel should be in relatively close proximity to the attractions. All of the information is recorded in the travel proposal document, as shown in Figure 2, and sent to a director for audit. But suppose that the principal cannot book Disneyland tickets after he has decided to go to Hongkong for travelling. He has to change either the travel date or the amusement. Alternatively, he can decide to visit Disneyland in California.

When an exception occurs in an execution process, it is essential that a rollback activity (a changed date, amusement, or destination) be found. In this example, when activity a9 throws the 'no available tickets' exception, there are three candidate activities to rollback to, which are a1, a2, and a5. If the system chooses a1 as the rollback activity, the exception can definitely be resolved. However, in this case, there will be many compensation processes, because the system will have to cancel many attempts. If the system chooses to rollback to activity a5, there are not so many activities to be undone, but it is also uncertain whether the system will find a suitable solution.

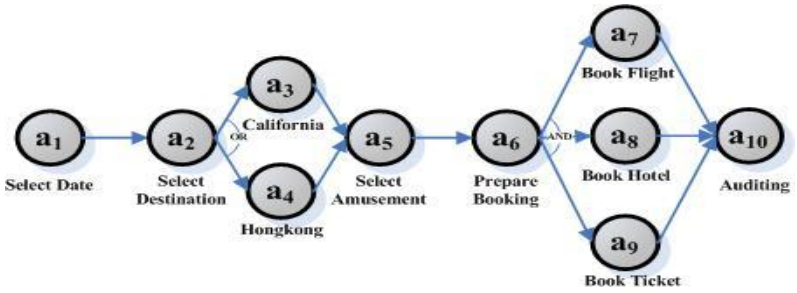

Figure 1: Travel plan process

Document Information		Process Information	
Document No.:	D00111	Process Name:	Travel Plan Process
Document Title:	Travel Proposal Document	Process Create Date:	2010.08.01
Document Create Date:	2010.08.05	Process Manager:	Tianming.Gan
Proposal Information			
1.Principal:	Tianming.Gan	2.Date:	
3.Destination:		4.Amusement:	
5.Hotel:		6.Flight:	
Cost Information			
7.Ticket Fees:		8.Hotel Fees:	
9.Flight Fees:		10.Total Fees:	
Proposal Suggestions			
11.Suggestions:			

Figure 2: Travel proposal document

Moreover, a document is revised many times while a process is executed. When dealing with exceptions, how to recover the document to a proper version to maintain consistency in the whole system is also a significant problem. In this paper, we propose a document-based recovery mechanism for provision of solutions to these problems.

2.2 Summary of Previous Work

Exception handling has been discussed in much of the previous research. A system that throws an exception can react either by terminating the execution process or by handling the thrown exception (Hagen and Alonso, 2000). Golani and Gal (2005) tried to handle exceptions by finding an alternative, replacement path for the failed execution path. They defined the rollback point as the nearest Xor-Split point to the exceptional point. However, an exception can occur for any one of several different reasons. Finding an alternative path to execute is not always the correct response. For example, when an online form fails to submit, the cause might be an uncompleted document or errors in a document. In some particular cases, the source of the problem might be an unstable network state. Nonetheless, dynamically finding a rollback activity in an executing business process is essential for recovery to a globally consistent state after failure.

Exception handling generally involves compensation flows. Compensation flows provide for process rollback after exceptions, as well as a set of compensating actions that leaves the process in a consistent state. However, these flows should be predefined. If no compensation process exists, the process operator probably accepts inconsistencies in which a completed activity is not voided.

Lerner, Christov, Osterweil, Bendraou, Kannengiesser and Wise (2010) describe several patterns of compensation. Eder and Liebhart (1998) provide a three-step mechanism to handle exceptions. The first step entails rollback based on the compensation type of activities in a workflow graph. In the next step, an agent determines whether to continue backward or to take an alternative path. The final step is forward execution, which might possibly lead to the same point of failure. The existing work of Do, Davis and Shan (1997) does not specify the stop point (i.e., rollback activity), implying that this point represents the decision on whether to continue. However, in many cases the parameter that drives this decision has been set before this point. Furthermore, these mechanisms are static (e.g., during build time) (Eder and Liebhart, 1998).

3. Preliminaries

In this section, we will introduce some basic definitions for business processes and documents. For better understanding of our approach, we will first provide definitions of the process structure model and the document structure model. Thereafter, based on these two definitions, we will explain document version and the three kinds of dependencies (Control dependency, operation dependency and document dependency). All of the work presented here is the necessary foundation of rollback activity finding and document recovery.

3.1 Process Structure Model, Activity State, and Process Execution Graph

In order to depict a business process, we need a model of its structure. The process structure model (PSM) described in Bae and Kim (2002) is useful for that purpose. The process structure is described as follows.

Definition 1 (Process Structure Model: PSM) A PSM is defined as a directed graph P that includes two sets A and L, which are a set of activities and a set of links, respectively. Therefore, a process p in P is a tuple (A, L):

□ A= {I | I =1, 2……, I} is a set of activities where I is the i-th activity and I is the total number of activities;

□ L ⊆ {(I, Ij)| Ij∈A, Ij∈A, and I ≠j} is a set of links where an element (I, Ij) represents the fact that I immediately precedes Ij.

When a process starts to execute, the state of activity will be changed. According to Aalst, Weske and Grünbauer (2005), there are four kinds of states representing the activity execution condition:

□ Defined: An activity has been defined in the PSM, but has not yet been executed.

□ Started: An activity has been triggered to execute and is being executed;

□ Committed: An activity has been executed without any interruption and successfully committed;

□ Aborted: An activity has stopped being executed due to some exceptional conditions having occurred. An activity in the aborted state will throw an exception to the system for exception handling.

Based on the above four states, to execute a process is considered to trigger the start point of the process, which is then called ‘started’. And to commit a process successfully is to trigger the end point of the process, which is then called ‘committed.’

In the BPMS, an activity does not take effect until it is committed to the system (Yu, Lie & Zang, 2009). We assume that the committing time is distinguishable. The process execution graph (PEG) is an acyclic non-branch graph used to depict the execution of a process in a run-time manner. The PEG (denoted as E) is a sequence of activities with state, which contains activities defined in PSM and their corresponding recovery activities (described in Section 5). Due to several uncertain factors in an open environment, it is possible to obtain several different PEGs while executing the same PSM many times.

3.2 Document Structure Model, Document Operation and Version

To manage the change of a document based on the execution of a business process, we also need a model for the structure of a document. The document structure model (DSM) described in Bae and Kim (2002) is useful for that purpose. A document is an ordered set of data gathered together in a certain organizational format for a certain communicative purpose. The DSM is described as follows.

Definition 2 (Document Structure Model: DSM) A document d is composed of a set of data fields as follows:

d = {fm | m=1, 2…, M}, where fm is the m-th data field and M is the number of fields in d.

A data field fm is specified according to a name/value pair. We use value (fm) to describe the value of the m-th field in document d.

In a business process, where document processing is the main task, execution is directly related to document handling. For users participating in a document-centric process, a mechanism by which a process is associated with a document is required. That is, for any activity in a process, the mechanism must determine the document fields to be dealt with. We use two sets to describe the document fields that an activity I deals with, which are denoted as a writing set W(I) and a reading set R(I). Both W(I) and R(I) consist of a set of document fields. For example, the reading and writing sets of activities in Figure 1 are shown in Table 1.

Table 1: Reading and writing sets

I	R(I)	W(I)
a1	∅	{f2}
a2	∅	{f3}
a3	∅	∅
a4	∅	∅
a5	{f3}	{f4}
a6	{f2, f3, f4}	∅
a7	{f2, f3}	{f6, f9, f10}
a8	{f2, f3, f4}	{f5, f8, f10}
a9	{f2, f4}	{f7, f10}
a10	{f1, f2, f3, f4, f5, f6, f7, f8, f9, f10}	{f11}

Further, we use the symbol Δ_w (I, fm, v, v') to describe the writing operation of activity I to field fm, which means that while executing I, operation Δ_w will write value v' to field fm with the original value v. Symbol ∅ is used to describe a field of no value.

A document change is detected automatically when a document modified by an activity is checked in (Bae and Kim, 2007). Modification of a document generates a new document state. A document undergoes several modifications through the activities of a process. Therefore, the history of changes needs to be managed systematically through repetitive modifications. Each document state is called a document version. Our system manages histories of document changes by using the following definitions of a version and a version graph.

Definition 3 (Document Version and Version Graph) Let d denote a document, and vp(d) the p-th version of document d. A Version Graph for d is a directed acyclic graph VG = (V, F), such that

V = {vp(d) | p =1,2, ..., P};

□ F = {(vp(d), vq(d)) | vp(d)∈V, vq(d)∈V, p≠q, and vq(d) = δ(vp(d))},where δ is a version-creation function. That is, vq(d) =δ(vp(d)) indicates that the q-th version of d is derived immediately from the p-th version of d.

Apparently, the reading set of an activity has no effect on the document version. A document version vp(d) can be changed by an activity I to vq(d), if and only if the writing field set of I is not empty: W(I)≠∅. Furthermore, if there is more than one writing field in W I), the version creation function δ is the combination of all of the writing operations to fields in W(I). We use the symbol to describe the combination of a set of operations:

$$\delta = \coprod_{\{f_m \mid f_m \in W(a_i)\}} \Delta_w(a_i, f_m, v, v') \tag{1}$$

$$v_q(d)=\delta(v_p(d)) = (\coprod_{\{f_m \mid f_m \in W(a_i)\}} \Delta_w(a_i, f_m, v, v'))v_p(d) \quad (2)$$

3.3 Dependencies

3.3.1 Control Dependency

Consider two arbitrary activities I and Ij; if the link $\{(I, Ij)\} \subset L$, we say that I is reachable to Ij. Moreover, we call this reach ability 'control dependency.' The notation $I \rightarrow Ij$ is used to indicate that Ij is control-dependent on I.

Control dependency $I \rightarrow Ij$ indicates that Ij is executed directly after I in the process. Dependency $\rightarrow$ is transitive and asymmetric. For example, in the process shown in Figure 1, we can obtain that a3 is executed after a1 from $a1 \rightarrow a2$ and $a2 \rightarrow a3$, which is denoted as $a1 \rightarrow a3$. Generally, we can use $I \rightarrow Ij$ to depict the fact that activity Ij must be executed after I, without considering whether there is an activity between I and Ij or not.

The dependency $\mapsto$ is a partial order because not every two arbitrary activities are reachable. Alternatively, we use notation $\|$ to represent that activity Ij is not reachable from I. Furthermore, there are two kinds of unreachable. One is parallel, which is denoted as $\|_P$. Notation $I \|_P Ij$ means that I and Ij can be executed in parallel, such as activity a7, a8, and a9 in Figure 1. The other kind of unreachable is mutual exclusion, denoted as $\|_E$. Looking at activities a3 and a4 in the example, the execution of a3 indicates that a4 cannot be executed.

3.3.2 Operation Dependency

Given dependency between two activities $I \rightarrow Ij$, we define the three operation dependencies as follows:

Read Operation Dependency:

If $(W(a_i) - \bigcup_{\{k \mid a_i \mapsto a_k \mapsto a_j\}} W(a_k)) \bigcap R(a_j) \neq \varnothing$, which means Ij reads some fields after I writes them. We call Ij a read operation dependent on I, which is denoted as $I \rightarrow Ij$;

Anti-read Operation Dependency:

If $R(a_i) \bigcap (W(a_j) - \bigcup_{\{k \mid a_i \mapsto a_k \mapsto a_j\}} W(a_k)) \neq \varnothing$, which means Ij modifies some fields after I reads them. We call Ij an anti-read operation dependent on I, which is denoted as $I \rightarrow Ij$;

Write Operation Dependency:

If $(W(a_i) - \bigcup_{\{k \mid a_i \mapsto a_k \mapsto a_j\}} W(a_k)) \bigcap W(a_j) \neq \varnothing$, which means Ij modifies some fields after I writes them. We call Ij a write operation dependent on I, which is denoted as Ij.

All of the operation-dependent relations ($\rightarrow_r$, $\rightarrow_a$, and $\rightarrow_w$)are intransitive. From the well known results of parallel computing, if an activity Ij is operation-dependent on another activity I, they cannot run concurrently, and Ij should be executed after executing I; otherwise, we will obtain erroneous results.

3.3.3 Document Dependency

Consider that some document fields' values are dependent on other fields' values, such as when one person's age is related to one's birthday. In this case, if an activity changes one's birthday in the document, the related age should also be modified. Otherwise, the document cannot maintain consistency after committing the activity. Hence, we define the document dependency as follows:

Suppose a field f in document; if $value(f)=\Gamma(value(f_1),...,value(f_n))$, where n=1, 2 …, we deem that field f is value-dependent on all of the fields fn for n=1, 2 …, which is defined as <f | f1, f2 …, fn>. The symbol is a function that can return the value of field f.

For example, in Figure 2, value (f10) =value (f7) +value (f8) +value (f9), where function is plus. So f10 is value-dependent on f7, f8 and f9: <f10 | f7, f8, f9>.

4. Rollback Activity

In this section, we will develop an algorithm for finding a rollback activity when a process fails to execute activities.

When an activity cannot commit its writing operations in a document to the system, the process has to be halted, and an exception is thrown at the activity. We regard this activity—the activity for which the exception is thrown—as the exceptional activity, denoted as aexcp. Due to all of the activities in the PSM being predefined in build time, which cannot be changed in run time, activity aexcp must be the result if the input variables (e.g., R (aexcp)) are defective or incorrect.

4.1 Bad Field Set

When the process engine estimates the environment information erroneously, or does not consider such bad situations, it will cause the process to generate some defective activities. The defective activities then generate or corrupt some incorrect document fields directly. In addition, the dependent relations among activities and document fields can further spread the defects to other document fields.

We denote Bf as a set of fields whose values are defective or incorrect. Thus, Bf should be initiated by adding all of the fields in R (aexcp). Moreover, bad fields stand a good chance to spread their errors to other fields. We identify corrupted document fields based on the following theorem.

Theorem 1 Assume that Bf is a bad field set, which is already known. The value of a document field f is incorrect if, and only if, any of the following conditions is true:

1) $\exists f' \in Bf, <f \mid f'>$;

2) $\exists f' \in Bf, \exists a \in E, f' \in R(a), f \in W(a)$;

3) $\exists f' \in Bf, \exists I, Ij \in E, I \rightarrow_r Ij, f' \in R(I), f \in W(Ij)$.

Proof： Rule 1 says that a bad field can spread its errors to the field whose value is generated from it. Rules 2 and 3 indicate that the activity that reads bad field has a great risk of generating defective fields.

4.2 Candidate Rollback Activities

In the case of an exception, undefined in advance, the process should rollback to an activity in the PEG, from which it can change some defective fields in Bf and support complete execution of the business process. We refer to such an activity as a candidate rollback activity. It is necessary to mention that, under normal conditions, there are a series of candidate rollback activities when an exception occurs at one activity. We use the set Ra to denote the set of candidate activities, which is identified by the following theorem.

Theorem 2 An activity I ($I \in E$ and $I \mapsto$ aexcp) is a candidate rollback activity if, and only if, either of the following conditions is true:

1) $I \rightarrow_r$ aexcp;

2) $\exists Ij \in Ra, I \rightarrow_r Ij$.

Proof：Both rules 1 and 2 mean that exceptions can be handled only by rolling back to the activity that affects aexcp directly or indirectly.

4.3 Algorithm for Finding Rollback Activity

As we mentioned previously, the farther a process rolls back, the greater is the possibility of fixing problems, but more time will be required to cancel and re-execute activities. So, the means by which the rollback activity is found looms as very important. We need to consider two decision variables: reparability and reactivity.

Reparability is the factor that describes the possibility of fixing problems. The more bad fields will be modified after rolling back, the greater will be the possibility of solving exceptions. Reactivity is the factor indicating the time consumed in handling exceptions. The more activities there are in Redo Set (discussed in Section 5.2), the more time is consumed in handling exceptions.

Let function count () be the number of elements in a set. For a candidate rollback activity I, we calculate the values of reparability and reactivity as follows:

$$Reparability(a_i) = \frac{count(Bf \bigcap \bigcup_{\{k \mid a_i \mapsto a_k \mapsto a_{excp}\}} W(a_k))}{count(Bf)} \times 100\% \tag{3}$$

$$Reactivity(a_i) = \frac{count(E) - count(RedoSet)}{count(E)} \times 100\% \tag{4}$$

Often, there is an inverse relationship between reparability and reactivity, where it is possible to increase one at the cost of reducing the other. For example, a candidate rollback activity can often increase its reparability by modifying more document fields, at the cost of increasing the number of redo activities, which will reduce the recovery system's reactivity.

Thus, the trade-off between inconsistency and time turns out to be the contradiction between reparability and reactivity. Based on Rijsbergen's (1979) effectiveness measure, for any candidate rollback activity ai, we combine its reparability and reactivity by using the harmonic mean of Reparability(ai) and Reactivity(ai), which is shown as follows:

$$F_\beta(a_i) = (1+\beta^2) \cdot \frac{Reparability(a_i) \cdot Reactivity(a_i)}{\beta^2 \cdot Reparability(a_i) + Reactivity(a_i)} \tag{5}$$

where β ($\beta \geqslant 0$) is a default value used to denote the relative weight between reparability and reactivity. $0 \leqslant \beta < 1$ indicates that the weights reparability is higher than the reactivity; $\beta > 1$ means that the measure puts more emphasis on reactivity than reparability. Generally, we use $\beta=1$, where reparability and reactivity are evenly weighted.

A desirable rollback activity should keep Fβ 's value as close to 1 as possible. Therefore, we can determine the rollback activity by computing Fβ for all of the candidate rollback activities in Ra.

The algorithm for finding rollback activity is described below:

Algorithm 1 Find rollback activity

Set Bf = R(aexcp)

for any field f in document do

if∃f' ∈ Bf such that <f|f'> then

Bf = Bf ∪ {f}

end if

if∃f' ∈ Bf,a∈ E such that f'∈R(a)
and f∈W(a) then

Bf = Bf ∪ {f}

end if

if∃f'∈Bf,ai,aj∈E such that ai $\rightarrow_r$ aj
and f' ∈R(ai) and f∈W(aj) then

Bf = Bf ∪ {f}

end if

end for

for any activity ai such that ai∈E
and ai↦aexcp do

if ai $\rightarrow_r$ aexcp then

Set Ra ={ai}

end if

if∃aj∈Ra,ai $\rightarrow_r$ aj then

a = Ra ∪ {ai}

end if

end for

for any activity ai that ai∈Ra do
Obtain its RedoSet by Algorithm 3
Compute the following equations

$$Reparability(a_i) = \frac{count(Bf \bigcap \bigcup_{\{k \mid a_i \mapsto a_k \mapsto a_{excp}\}} W(a_k))}{count(Bf)} \times 100\%$$

$$Reactivity(a_i) = \frac{count(E) - count(RedoSet)}{count(E)} \times 100\%$$

$$F_\beta(a_i) = (1+\beta^2) \cdot \frac{Reparability(a_i) \cdot Reactivity(a_i)}{\beta^2 \cdot Reparability(a_i) + Reactivity(a_i)}$$

end for

Set aback as the activity ai where
Fβ(ai) is the closest to 1
return aback

5. Recovery Activities

In this section, we describe the issues relating to finding undo and redo activities in details. Also, we will continue to discuss the execution order rules between them when handling exceptions.

In this paper, we assume that if an activity a is defective, we can remove its effects in the execution process by invoking an activity Undo (a). To recover defective activities, we need to re-execute them. We denote the re-execution of activity a by Redo(a). Activities a and Redo(a) are different executions of the same task. Redo(a) refers to the execution when carrying out exception handling. Both redo activities and undo activities are considered as recovery activities.

5.1 Undo Activity

Definition 4 (Undo Activity) The undo activity Undo(ai) should satisfy all of the following 4 rules:

ai $\in$ E and the state of ai is committed;

ai is aexcp or aexcp ai aback;

$$W(ai) = W(Undo(ai)) \neq \emptyset;$$

$$\forall fm \in W(ai),\ ^{\Delta_w}(Undo(ai),fm,v',v) = {}^{\Delta_w^{-1}}(ai,fm,v,v')$$

Based on Definition 4, we can easily find that any activity ai between aback and aexcp (containing aback) can generate its undo recovery activity Undo (ai) by reverse writing operations in the document. However, it does not need to cancel all of the activities between aback and aexcp. In other words, if the process throws an exception at aexcp, and decides to rollback to aback for recovery, we should recover the document to a consistent version while the process is rolling back to maintain consistency in the system. Whereas, for those activities those have not made any modifications to the document or have demonstrably performed in the correct manner, it is inefficient for the system to cancel their effects and re-execute. Hence, we use activity set UndoSet to depict the activities whose impaction should be called off. As well, we develop the theorem and algorithm for generating UndoSet as follows.

Theorem 3： The undo activity Undo (ai) of activity ai should be put into UndoSet if, and only if, any of the following conditions are satisfied:

ai = aback;

$\exists$aj $\in$ A such that ai $\|_E$ aj and Redo(aexcp) $\mapsto$ aj;

$\exists$Undo(aj) $\in$ UndoSet such that aj ai and Redo(aexcp) $\mapsto$ ai;

∃Undo(aj)∈UndoSet, ∃fx, fy∈d, such that fy∈W(aj), fx∈R(ai), and <fx | fy>.

Proof: The objective of undo activities is to remove the defective data. Rule 1 indicates that a rollback activity should be undone. According to Theorem 2, any candidate rollback activity must affect aexcp directly or indirectly, so aback should remove its effect. Rule 2 means that an activity that has updated a document and will not be re-executed should be undone. Rules 3 and 4 dictate that after undoing an activity, the directly and indirectly affected activities should also be undone.

The algorithm for finding UndoSet is described below:

Algorithm 2 Find UndoSet of aback:

Contents: E' is used to denote the;

New semantic PEG by analyzing;

The PSM in advance:

Set UndoSet = {Undo(aback)}

ty ai such that ai∈E

and ai∉E' do

if W(ai)≠∅ and the state of ai is

committed then

UndoSet = UndoSet ∪ {Undo(ai)}

end if

end for

for all activity aj such that

Undo(aj)∈ UndoSet do

for all activity ai such that ai∈E

and ai∈E'do

if Undo(ai) ∉ UndoSet then

if the state of ai is committed

and aj $\rightarrow_r$ ai then

UndoSet = UndoSet ∪{Undo(ai)}

else if ∃fy∈W(aj) and ∃fx∈R(ai)

and<fx|fy> then

UndoSet = UndoSet ∪ {Undo(ai)}

end if

end if

end for

end for

return UndoSet

5.2 Redo Activity

Definition 5 (Redo Activity) For any activity ai, the redo activity Redo(ai) should satisfy all of the following 3 rules:

1) Undo(ai) UndoSet

2) R(Redo(ai)) = R(ai)

3) W(Redo(ai)) = W(ai).

Based on Definition 5, we can obtain that the redo activity must be undone first. Any activity that has not had its effects cancelled cannot be re-executed. Furthermore, the reading and writing sets of a redo activity should be the same as the activity itself. However, not all of the activities in UndoSet should be re-executed. Theorem 4 will talk about what kind of activity should be redone.

Theorem 4 Any redo activity Redo(ai) of activity ai should be put into RedoSet if, and only if, any of the following conditions are satisfied:

1) ai = aback;

2) $\nexists$aj $\in$ A, such that ai $\|_E$ aj and Undo(ai) $\in$ UndoSet;

3) $\forall$aj $\|_E$ ai, such that assumption (Redo(aback) $\mapsto$ aj) is not true and Undo(ai) $\in$ UndoSet;

4) $\exists$Undo(aj) $\in$ UndoSet, aj $\rightarrow_r$ ai;

5) $\exists$Undo(aj) $\in$ UndoSet, $\exists$fx, fy d, such that fy $\in$ W(aj), fx R(ai), and <fx | fy>.

Proof: As in the proof of Theorem 3, rule 1 indicates that a rollback activity should be redone. Rule 2 reflects the condition that activity ai is in all of the execution paths of the PSM. Rule 3 is derived from the condition that the process selects activity ai as a retry. Both rules 4 and 5 mean that when cancelling the effects of an activity aj, the reading value of ai will be affected; thus ai should be redone to update its input data.

the algorithm for finding RedoSet is described below:

Algorithm 3 Find RedoSet of aback

Set RedoSet={Redo(aback)}

for all activity ai such that

Undo(ai) $\in$ UndoSet do

if ai $\in$ E' then

RedoSet = RedoSet $\cup$ {Redo(ai)}

end if

for all activity aj that aj $\in$ E do

if Redo(aj)∈RedoSet then
break
else if W(ai)∩R(aj)≠∅ then
RedoSet = RedoSet ∪ {Redo(aj)}
else if ∃fy∈W(aj) and ∃fx∈R(ai)
and<fx|fy> then
RedoSet=RedoSet ∪ {Redo(aj)}
end if
end for
end for
return RedoSet

6. Conclusion

For the purpose of performance improvement, the analysis and the mitigation of impulse noise in multi-carrier communication are getting more and more urgent. Weibull distributed impulse noise is a credible mathematical model that is verified by measured signals, so it can be applied in theorem analysis. According to the randomness of impulse noise, the correlation coefficient of the impulse noise power and the distribution probability are achieved through strict theoretical derivation. And the simulation results prove the reliability of the theory derived by this paper. Under the analysis of the probability distribution of correlation coefficient, a relationship between the severities of impulse noise to the location of the peak of probability is found. This relationship can serve as a kind of new method to estimate the characteristic of the impulse noise, and have a certain potential application.

References

Bae, H, & Kim, M. (2007). Process based storing and reconstructing of XML form documents. *Computers in Industry*, 58(1), 87-94.

Bae, H., & Kim, Y. (2002). A document-process association model for workflow management. *Computers in Industry*, 47(2), 139-154.

Do, W., Davis, & J. Shan, M. (1997). Flexible specification of workflow compensation scopes. GROUP '97 Proceedings of *the international ACM SIGGROUP conference* on supporting group work: the integration challenge, 309-316, Phoenix, November 1997.

Eder, J., & Liebhart, W. (1998). Contributions to exception handling in workflow management. *Proceedings EDBT Workshop on Workflow Management Systems*, 3-10.

Golani, M., & Gal, A. (2005). Flexible business process management using forward stepping and alternative paths. *Third International Conference on Business Process Management*, 48-63.

Hagen, C., & Alonso, G. (2000). Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10), 943–958.

Lerner, B. S., Christov, S., Osterweil, L. J., Bendraou, R., Kannengiesser, U., & Wise, A. (2010).Execption Handling Patterns for Process Modeling. *IEEE Transactions on Software Engineering*, 36(2), 162-183.

van der Aalst, W. M. P., Weske, M., & Grünbauer, D. (2005). Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2), 129–162.

van Rijsbergen, C. J. (1979). *Information retrieval (2nd ed.)*. London: Kluwer Academic Publishers.

Wang, M., & Sun, Z. (2010). Handbook of Research on Complex Dynamic Process Management: Techniques for Adaptability in Turbulent Environments. *Economics and Finances*, 63.

Yu, M., Lie, P., & Zang, W. (2009).The implementation and evaluation of a recovery system for workflows. *Journal of Network and Computer Applications*, 32(1), 158–183.