Optimizing Exception Handling in Workflows using Activity Restructuring

Mati Golani[1], Avigdor Gal[2]

[1]Software engineering department, Ort Braude College, Israel
iemati@ie.technion.ac.il

[2] Technion - Israel Institute of Technology
avigal@ie.technion.ac.il

Abstract. Exception handling is the process by which a failure in a process is mitigated. Depending on the specifics of an exception, exception handling may range from halting a process, through attempts of activity reactivation, to an identification of an alternative path to successful completion of a process. Designing efficient exception handlers is not a simple task. By their very nature, exceptions are rare events which may result in poor design of exception handlers in terms of cost and logic. In this work we aim at improving exception handling performance in workflow management systems (WfMSs), a task which has been recognized as a fundamental component of WfMSs that is critical to their successful deployment in real-world scenarios. Our approach is based on the observation that when designing a business process as a workflow, a designer has some degree of freedom in streamlining actions. Therefore, we propose workflow restructuring as a main tool in reducing the cost of exception handling. We believe that restructuring of a workflow, based on exception efficiency consideration, can increase the overall productivity of the business process. Although the rarity of exceptions allows amortizing their costs over time we cannot ignore exception costs altogether. Therefore, we use a cost-based approach to prioritize their impact on the workflow design. Our main contribution is the provision of a methodology for exception handling optimization at the workflow design phase.

1 Introduction

Exception handling is the process by which a failure in a process is mitigated. Depending on the specifics of an exception, exception handling may range from halting a process, through attempts of activity reactivation, to an identification of an alternative path to successful completion of a process. Exception handling is a crucial component in an efficient process management [1, 4], and has great impact on any system performance.

Designing efficient exception handlers is not a simple task. By their very nature, exceptions are rare events which do not enjoy the advantages of common processes, which are easily programmed with much expert information injected

into them. Thus, exception handlers may well be ill-designed, affecting both the correctness and the efficiency of the process. Avoiding their design altogether is also not a valid option. During runtime, process operators see only a narrow perspective of the process, and given an exception, will not have sufficient information to effectively manage it. In those cases in which the operator mitigates the exception, the solution nay be neither optimal nor effective.

Another aspect of exception handling involves their overwhelming amount with respect to the "normal" process size. Clearly, for any "right" way of performing a process, there may be many things that could go wrong, each possibly requiring its own exception handling. Therefore, the modeling of many exception handlers, combined with the lack of expert support (due to the rarity of exceptions) would likely result in a poor design of exception handlers, both in terms of logic and in terms of cost.

In this work we aim at improving exception handling performance in workflow management systems (WfMSs). WfMSs nowadays serve as the main process-based technology in enterprise environment, evolving in the past 10-15 years to support the design and execution of business processes. Their fundamental approach is to separate the description of the process flow from the application functions at design time. Workflows define a business process in terms of activities (also called actions or tasks). Activities, together with temporal constraints on execution ordering define a business process [9]. Efficient exception handling has been recognized as a fundamental component of WfMSs that is critical to their successful deployment in real-world scenarios [1].

Our approach is based on three basic observations. First, we observe that when designing a business process as a workflow, a designer has some degree of freedom in streamlining activities, and the decision on one specific design is typically based on organizational practices. Therefore, we propose workflow restructuring as a main tool in reducing the cost of exception handling. Secondly, we observe that exception handling design is typically performed only after the "normal" process of execution has been designed. Clearly, current methodologies of process design are by no means geared towards exception handling. Therefore, we believe that restructuring of a workflow, based on exception efficiency consideration, can increase the overall productivity of the business process. Nevertheless, we do not consider exceptions to be "first-class citizens." Our proposed methodology assumes that process design is indeed done first, followed by exception handler design and possible restructuring of the original workflow design. Thirdly, it is clear that the rarity of exception requires amortizing their costs over time. This, however, is not parallel to ignoring exception costs altogether. Therefore, we use a cost-based approach to prioritize their impact on the workflow design.

We next present an example to illustrate our approach in a given scenario.

Example 1. (**Workflow process example**) Consider a process that handles package tour reservations (see Figure 1). This process uses some local applications (member deals), as well as Web services (hotel registration: activity 0), with some special offers available to Gold members only (activities $5 - 8$). We
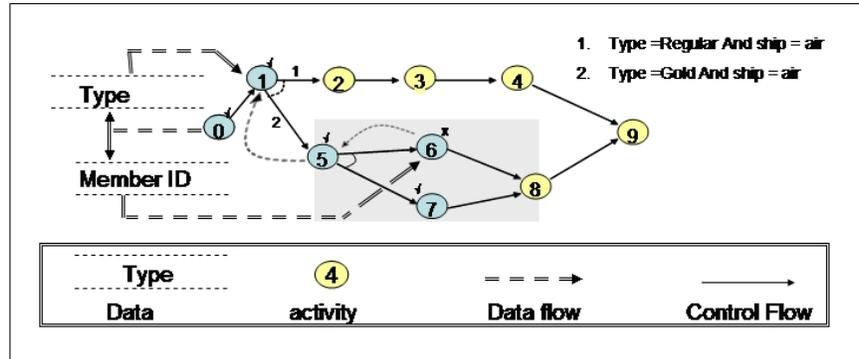
Fig. 1. An example of a business process

consider an example that involves a failure of the special offer system for a Gold customer (activity 6). Consider an exception in which the system chooses an alternative path [4] that allows successful completion of the business process while bypassing the special offers. In Figure 1, activities $0, 1, 5,$ and 7 were performed for this workflow instance, yet a failure at activity 6 blocks the process and prevents its completion. The exception handler for this failure takes an alternative paths to the current path, rollback activities $6, 7, 5,$ and 1, and re-execute the process from activity 0.

One should consider the costs of such operations. By rollback an activity, there may be a penalty cost during the compensation activity execution (activities $6, 7, 5, 1$ in the above mention example). Similarly, re-execution of a previously executed activity may also has its extra costs (subset of activities 0,and 1 in the given example). By reducing the number of compensated activities, and re-executed activities in the execution path of the exception handler, the total cost of the exception handler execution may be reduced.

Our specific contribution is twofold. First, we provide a methodology for exception handling optimization at the process design phase. Second, we formally define an exception handler as a workflow and discuss its combination with existing workflows. The rest of the paper is organized as follows: In Section 2 we present the workflow graph-based model, including exception handlers structure, and association dependencies between activities. We also discuss the details of our cost model and present the optimization problem of exception handling design. In Section 4 we present a the exception handler optimization methodology, starting from a single exception handler, and extending it to multiple exception handlers. We conclude in Section 6 with a short discussion of current achievements and future work.

2   Workflow Model

In this section we provide background information, and present the various elements of our model, based on ADEPT WSM nets [6]. ADEPT provides a natural mechanism for dynamic workflow restructuring. It combines a graphical representation with a solid formal foundation, taken from graph theory, allowing both reasoning and an execution engine [7]. Similar models have been presented elsewhere already (*e.g.*, [8, 4]), so this section is brought here for expository completeness only.

A workflow model can be described as a graph (ADEPT WSM net) $G(V, E)$ ($V = (V_a \cup V_d)$; $E = (E_c \cup E_d \cup E_s)$), where V_a is a set of activities, V_d is a set of data parameters, E_c is a set of control edges, E_d is a set of data edges, and E_s is a set of synchronizing edges. Synchronizing edge are quite useful, and allow activities on parallel threads to synchronized. We will use this functionality in Section 4.3.

Control edges reflect temporal constraints on the ordering of activities. An ordering of (a_i, a_j) may reflect that a_j requires the input of a_i for its processing. Alternatively, if a_i and a_j make use of a common, limited resource, a designer may decide to avoid collisions by serializing their activation. Finally, such an ordering reflects the existing business process modeled by the workflow. Recall that one of our main observations is that when designing a business process as a workflow, a designer has some degree of freedom in streamlining activities. Thus, we propose workflow restructuring by possibly changing existing ordering in a workflow. Clearly, not all orderings are subject to change. Therefore, we introduce a new function, $TG : V_a \times V_a \rightarrow \{none, prec, tight\}$, which takes two activities a_i, a_j and returns their tightness status that can be one of three values. *none* represents no tightness constraints, *prec* requires that a_i precedes a_j and *tight* requires that a_i immediately precedes a_j (*i.e.*, $e_{ij} \in E_c$). We use a matrix notation TG_{ij} to represent $TG(a_i, a_j)$. The default value of TG_{ij} is none, allowing restructuring of activities without any restrictions. We do not expect the designer to provide a full TG description. Rather, in Section 4 we introduce an iterative exception handler optimization algorithm in which more tightness constraints are added after each iteration, and the restructuring is modified accordingly. Since the initial TG matrix set by the designer (can be initiated with *none* values), we may infer some extra knowledge to TG. Recall that activity a_i which sets a parameter value which is read by another activity a_j, define a precedence relation (i.e. $TG(a_i, a_j) = prec$). Another issue is xor splits. Xor splits provide two or more alternatives of which each executes different activities (until reaching the corresponding join). Therefore, it make sense that activities within the xor block should not be propagated out of this block's scope because they should not be executed in all cases. Thus, for such a xor split activity a_i and any activity a_j within the xor block we may also set $TG(a_i, a_j) = prec$.

Given a workflow graph $G(V, E)$ and an activity $a \in V_a$, a *Xor split point of a* is a Xor split activity a_i with a Xor join activity a_j such that a_i is a predecessor of a and a_j is a successor of a. *The Nearest Xor split point of a* (denoted $NXSP(a)$) is a Xor split point of a, a_i, which satisfies that any other
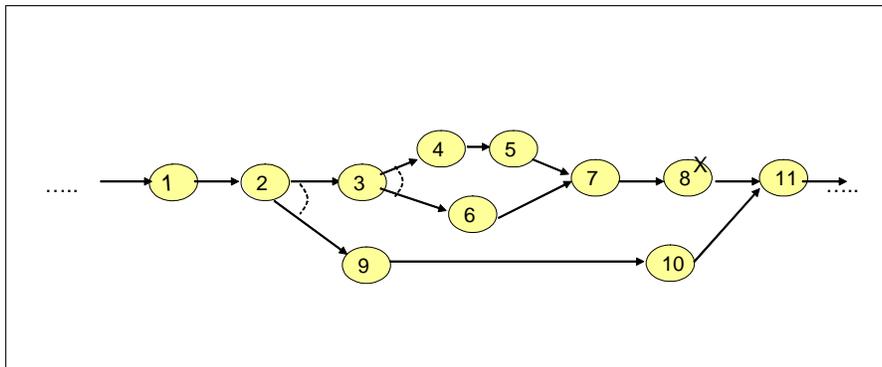
Fig. 2. An illustrative section from a generic process.

Xor split point of a, a_j is also a Xor split point of a_i. *And split point* and *Nearest And Split Point (NASP)* are similarly defined.

An *alternative path* to activity a_i is a path of execution that avoids a_i. We use well structured processes as an input for our model (or use a normalization algorithm in order to transform the process to a well structured one [3]). This property matches the WFMC definition (in interface 1) of *full-blocked workflows*. An alternative path will usually start with a xor split activity. It is worth mentioning that not all predecessor xor split activities may serve as a starting point for an alternative path. Note activity 3 in Figure 2. due to a failure in activity 8, activity 3 may not be defined as the starting point of the alternative path since it leads directly back to the failing activity 3. Activity 2 on the other hand may function as an alternative path source.

Given a workflow graph $G(V, E)$, $Inst(G)$ represents an instance of G. $Inst(G)$ encapsulates instance-related data, such as activity state and input/output parameter values. $Inst(G)$ is a DAG and loop constructs in $G(V, E)$ are removed by duplicating loop blocks and re-labeling of activities.

An activity in $Inst(G)$ can be classified into one of the following states: uninitiated (yet, but on an execution path), void (on path that was not invoked), completed (finished on current path), compensated, or failed.

Example 2. In the example given in Figure 1 $V_a = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $V_d = \{type, MemberID\}$, $E_c = \{$0-1,1-2,2-3,3-4,4-9,1-5,5-6,5-7,6-8,7-8,8-9$\}$, $E_d = \{0-Type, Type-1, 0-MemberID, MemberID-6\}$. $NASP(6) = 5$, and $NXSP(6) = 1$. $Inst(G)$ can contain the following additional data: $Type = "Gold"$, $MemberID = "14352"$.

3 Exception Handling

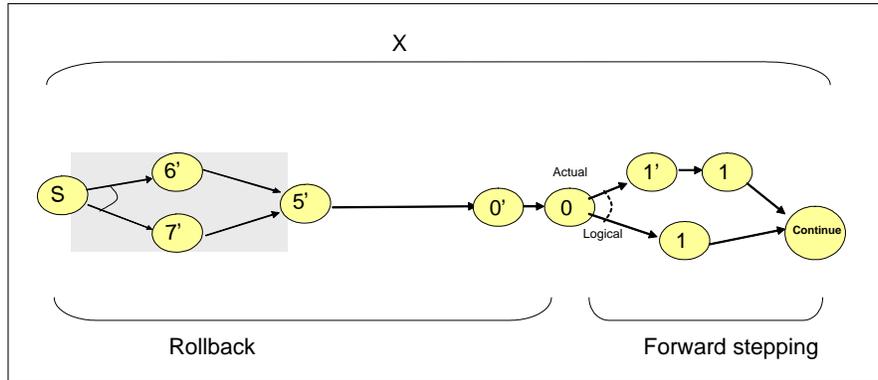In this section we provide the exception handling model, and present a cost model for exception handlers.

Fig. 3. An exception handler example

3.1 Exception Handling Model

We define an *exception handler* to be a workflow $X(V_X, E_X)$, executed in response to an occurrence of an exception for which it was defined. Given a workflow $G(V, E)$ and an exception handler $X(V_X, E_X)$, we define an operator *Apply* such that by applying $X(V_X, E_X)$ to $G(V, E)$ one receives a revised workflow model

$$G'(V', E') = Apply(G(V, E), X(V_X, E_X), v_s, V_e),$$

where $\{v_s\} \cup V_e \subseteq V$. v_s specifies the failing node in G and V_e is a set of nodes in G from which the normal operation of G will resume. Therefore, in G' there will be an edge $(v_s, root(X))$ connecting the failing activity with the exception handler. Also, any edge $(v_1, v_2) \in G'$ such that $v_1 \in V_X$ constrains v_1 to be in V_e. The exception handler structure is devided into two sections (See Figure 3), a *rollback* section that executes compensating activities that are apriori known as required to be executed, and a *forward stepping* section that executes the activities *logically* or *actually* (to be defined in Section 3.2).

Each exception handler can be reflected in its corrsponding process with three reference points:

- *Start point a_{s1}* - the activity where the exception occured.
- *Stop point a_{s2}* - the activity where the control is returned to the original process
- *Target point a_t* - the activity where the roll back section ends.

Example 3. (**Exception Handler**) Figure 3 defines an exception handler to an exception in activity 6 of Figure 1. The semantics of these activities will soon be discussed. 4 presents the combined result of applying the exception handler X of Figure 1 on the workflow model of Figure 1. The dotted edges in Figure 4 represent the shift of control from the original process to the exception handler and back to the process, once the exception handler execution is completed.
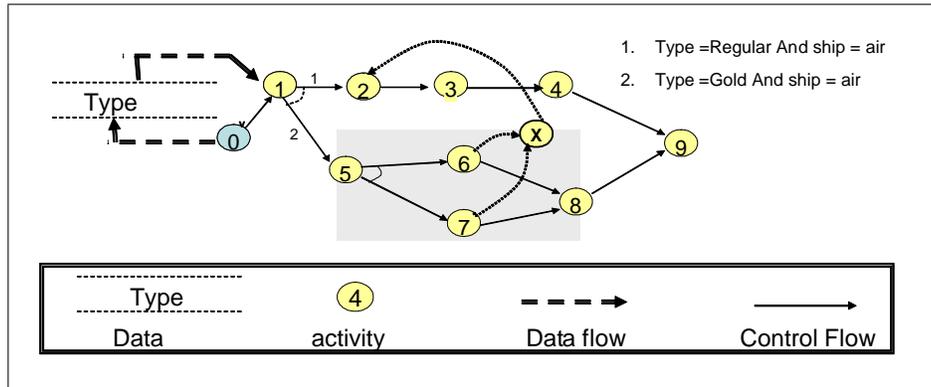
Fig. 4. Exception handler applied to the process

It is worth noting that the exception handler x starting point servse as a Xor join closing the block started at activity 5 (while voiding the control edges leading to activity 8), and the flow of control is returned to the process at activity 2 which serves as a Xor join of the block started at activity 1. This modificaton to the process still keeps the modified process normalized.

There are three types of activities an exception handler can use. First, it can use activities in the workflow, some of which may be re-activated, while others may be activated for the first time for a given workflow instance. The second type is that of *compensation* activities, also known as *undo* activities and *semantic rollback* activities [5, 2]. A compensating activity needs to be pre-defined, is associated with a single or combined set of workflow activities and is typically used for reversing the impact of activities that were already performed for a given instance. Lastly, an exception handler can use activities that are not defined in the workflow altogether. Looking at Figure 3 the rollback section is executing the compensating action of activities 6, 7, 5, 1,and 0. Later in this exception handler, activity 0 is re-executed, and activity 1 can be executed, or just reported to be executed (i.e. *Logical*) to the workflow system.

To illustrate the notion of exception handling further, we now present two types of exception handlers. The proposed algorithm is not limited to these types, though. We denote the first type a *repeat execution exception handler*. Such an exception handler attempts to repeat a subgraph of a workflow model by first applying compensating activities to the part that was already executed, followed by re-executed activities. A repeat execution exception handler for the same failure in activity 6 is presented in Figure 5. This exception handler rollbacks only activities which may be required to be executed again. In this case we would like to try to re-execute a failing activity 6 with a different input value in parameter $MemberID$. By re-executing activity 0, and activity 1 (which is dependent on activity 0 output), the process may be able to re-execute activity 6 successfully. Note that activity 5 as not been compensated nor re-executed
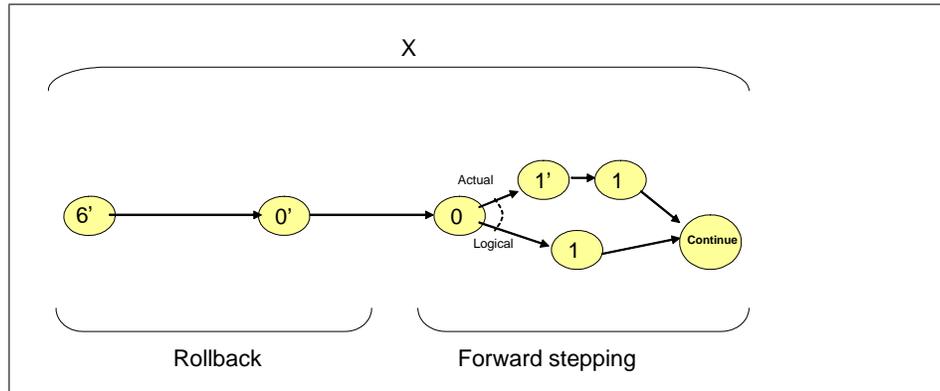
Fig. 5. An example for repeat execution exception handler

since it completed successfully and it is not dependent on any of the probably modified parameters $MemberID$, and $Type$.

The second type is denoted an *alternative path exception handler* which was first introduced in [4]. This type of an exception handler identifies an alternative path in the workflow graph that avoids the failed activity. Alternative paths are paths in a workflow graph with a mutual prefix and a different suffix that starts at a mutual Xor split point. Alternative path exception handlers combine the use of compensating activities, re-execution of activities and first-time activity activation. In Example 3, activity 5 for example is compensated, Activity 1 is compensated and re-executed, and activity 2 will be executed for the first time once this exception handler is completed.

We next draw the reader's attention to the special cases of exception handlers.

- $a_{s1}{=}a_{s2}{=}a_t$ Degenerated exception handler. This is actually a re-execution of the failing activity.
- $a_{s2}{=}a_t$. This exception handler executed the rollback section of compensating activity with no logical execution or re-execution of activities.
- $a_{s1}{=}a_t$. This exception handler compensates only the failing activity and begins the execution of alternative activities.
- $a_{s2}.type{=}$ *Xor split*. This exception handler implements the alternative path approach (unless the failing activity itself is the Xor split activity).

The construction of exception handlers is beyond the scope of this paper. The interested reader is referred to [3]. In what follows we assume that a designer provides us with an exception handler to be optimized.

3.2 Cost Model

To establish a notion of optimality, we next discuss the various components of a cost model of an exception handler. To start with, there may be an initiation

cost for starting a new exception instance. We assume that this cost is identical for all exception handlers and therefore we shall avoid discussing it further.

Given an activity a, we denote by $C(a)$ the cost of activating a. This cost may be in terms of time or monetary value. We do not differentiate activation from re-activation. Also, given a', a compensating activity of a, $C(a')$ is similarly defined to be the cost of compensating a using a'.

There are two modes of activity execution in an exception handler, namely *actual* and *logical*. An actual activation of an activity a involves the invocation of a routine associated with a or the addition of a work item to an item list of some role in the organization. A logical activation of a requires only recording its activation in the WfMS without actually activating it. Logical activation can be applied whenever the outcome of a previous activation of a within the same instance can be utilized without any modification in the exception handler. For example, some activities can be assumed to yield the same output whenever its input values are the same. As another example, activities may be associated with a *valid time*, which implies that the same activity with the same input values will provide the same within its valid time. As a concrete example, consider a medical treatment process that provides a dosage of medicine which is based on the results of a lab test. The activity's input is the *patient id*. If an exception occurs, the process may be required to rollback and provide a different dosage, or another medicine. The lab test activity can be assumed to remain valid within a certain time, after which it should be performed again.

In what follows, $C(a)$ is set to 0 whenever a requires only a logical activation. If an actual activation is involved, then $C(a)$ is assigned with its full cost. It is worth noting that in some cases, it is known in advance whether an actual or a logical activation is required, *e.g.*, whenever the exception handler is guaranteed not to change the input values, and no valid time is associated with the activity. In other cases, however, the activation type may only be known in real-time.

Finally, we assume costs are cumulative. Therefore, we define the cost of exception handler X to be the sum of costs of all activities in X. Formally,

$$C(X) = \sum_{a \in X} C(a) \tag{1}$$

4 Exception Handler Optimization

This section introduces a design time methodology for optimizing exception handler execution by avoiding the redundant execution of tasks. We start in Section 4.1, and introduce the problem. We then continue in Section 4.2 by presenting the approach for this optimization via an example. In Section 4.3 we present our proposed solution for a single exception handler. Then, in section 5, we discuss extending the solution to the case of multiple exception handlers.

4.1 Problem Definition

We are now ready to formally introduce our problem. Let G a workflow graph and X_i be a set of exception handlers, each with a frequency f_i (where frequency

can either measure how likely this exception handler is to be used or reflect an importance a designer assigns with this exception handler). We require that for all i $0 \leq f_i \leq 1$ and that $\sum_i f_i = 1$. Each X_i is composed of a set of execution activities $a_{i,j}$, each associated with cost $C(a_{ij})$, and the cost of X_i is computed according to Eq. 1. We would like to restructure G so as to minimize the weighted cost of exception handling, subject to tightness constraints. Formally put,

$$
\begin{aligned}
Min & \sum_{X_i} f_i C(X_i) \\
s.t. \quad & TG_{ij} = prec \Longrightarrow \text{there is a path from } a_i \text{ to } a_j \text{ in } G \\
& TG_{ij} = tight \Longrightarrow e_{ij} \in E_c
\end{aligned}
\tag{2}
$$

It is worth noting that tightness constraints may not be known in advance. Rather, we consider the optimization process to be an iterative process, in which a designer is presented with proposed design restructuring, and in response to additional tightness constraint, the process of redesigning is initiated once more.

4.2 Exception Handler Optimization Methodology

To illustrate our approach, consider the process presented in Figure 6. The exception handler for activity 6 failure takes an alternative path from activity 3 and follows edge $e_{3,12}$. An alternative path would follow the path that goes through activities 3 and 4 rather than activities 3 and 12. To allow this change, the condition $C1$ should be modified to "true." This entails making changes to the value of C, which was last updated at activity 1. Therefore, the exception handler should rollback to activity 1, compensating activities 3, 2, and 1 and then perform (either actual of logical) re-execution of activities 1, 2, and 3. The exception handler graph would look like $6' \rightarrow 5' \rightarrow 4' \rightarrow 3' \rightarrow 2' \rightarrow 1' \rightarrow 1 \rightarrow 2 \rightarrow 3$, where the execution of activity 2 can be *logical* or *actual* according to the circumstances. We can reconstruct the process and reduce the exception handler costs by reducing the execution part which is before the xor split as well as the execution section after the xor split. By shifting activity 1 forward after activity 2 we reduce the requirement for compensation (and/or re-execution) of activity 2 (which belongs to the pre xor split section). The result would be $6' \rightarrow 5' \rightarrow 4' \rightarrow 3' \rightarrow 1' \rightarrow 1 \rightarrow 3$.

Using the same rational, and looking at activity 8, an exception handler to deal with activity 8 failure would have to compensate activity 12. The exception handler would have the following structure $8' \rightarrow 12' \rightarrow 3' \rightarrow 2' \rightarrow 1' \rightarrow 1 \rightarrow 2 \rightarrow 3$. By propagating activity 8 before activity 12 we may also reduce the exception handler cost (post xor split section), resulting with the following exception handler (including the modification to the pre xor split section) $8' \rightarrow 3' \rightarrow 1' \rightarrow 1 \rightarrow 3$.

4.3 Activity Propagation

Recall that our exception handling approach is based on rollback to a previously executed activity, and in many case it is required to re-execute. By restructuring
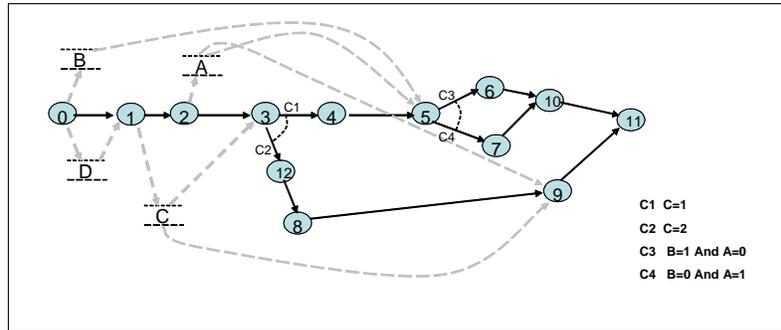
Fig. 6. An example process for propagation

the process, we may be able to shorten the rollback process, minimizing the number of rolled-back activities,and re-executed activities, in order to execute the exception handler more efficiently. In order to minimize the number of rolled-back activities, we could reconstruct the graph structure so that the rollback target activity is propagated as close as possible to the exception handler stop point activity.

As stated before, the target activity is the prime activity that modifies the parameter which is responsible for the satisfying change. Each such modifying activity (including the prime one) will have its compensating activity in the roll-back section of the exception handler, and is a candidate for propagation. Each candidate activity is first verified that it does not have a mandatory precedence relation with its successor (see section 2). Those candidate activities that pass the verification are propagated until it either reaches the stop point activity, or it reaches a dependent activity that does not allow the switch between them.

Given the example in Figure 6, and an exception handler X that consist of compensating activities $4' - 3' - 2' - 1' - 1 - 2 - 3$, using an alternative path starting from activity 3 leading to activity 8. $a_{s1} = 4$, $a_{s2} = 3$,$a_t = 1$. For each compensating activity a' in X between a_t and a_{s2}, we will try to propagate its corresponding activity a in G (activity 1 in the example), until reaching the stop point activity a_{s2}(activity 3 in the example). In between there is only activity 2 that does not have a mandatory precedence relations with activity 1. For each compensating activity a' in X between a_{s1} and a_t, we will try to back-propagate the failing activiy a_x before a' corresponding activity a in G. The revised process is presented in Figure 7. The Algorithm is presented in Algorithm 1. The new and shorter exception handler for this process consist of the following execution $4' - 3' - 1' - 1 - 3$

It is worth mentioning that Algorithm 1 is näive in the sense that it refers to a relaxed case of singular threads of propagation excluding splits and joins. In order to consider such more complex structures, the algorithm can be revised so that instead of iterating over activities one at a time (Lines 15-19), once reaching a block, the algorithm checks the dependencies of **all** block members with the

---

Algorithm 1 activities propagation algorithm

1: **Input:** Graph G, TG, E-exception handler.
2: **Output:** revised Graph G'
3: Process:
4: go over E end detect the start point a_1, stop point a_2, and target point a_t.
5: //post xor split section
6: $a_x = a_1.predecessor$
7: **while** $a_x \neq a_2$ **do**
8:    **if** $TG[a_x, a_1] \neq none$ **then**
9:      Break
10:    **else**
11:      **if** $TG[a_x.predecessor, a_x] \neq tight$ **then**
12:        shift activiy a_1 before a_x in G
13:        $a_x = a_1.predecessor$
14:      **else**
       $a_x = a_x.predecessor$
15:      **end if**
16:    **end if**
17: **end while**
18: //pre xor split section
19: make a reverse BFS traversal starting from a_2, and insert visited activities that write parameters read by a_2 to vector V
20: **for** each $a_i \in V$ **do**
21:    **if** $\exists e_{i,m} | TG[a_i, a_m] \in tight, prec$ **then**
22:      continue //to the next activity to propagate
23:    **end if**
24:    $a_k = a_i.successor$
25:    **while** $a_k \neq a_t$ **do**
26:      **if** $TG[a_i, a_k] = none$ **then**
27:        **if** $TG[a_k, a_k.succesor] \neq tight$ **then**
28:          propagate activity a_i after activity a_k in G
29:        **else**
         $a_k = a_k.succesor$
30:        **end if**
31:      **end if**
32:    **end while**
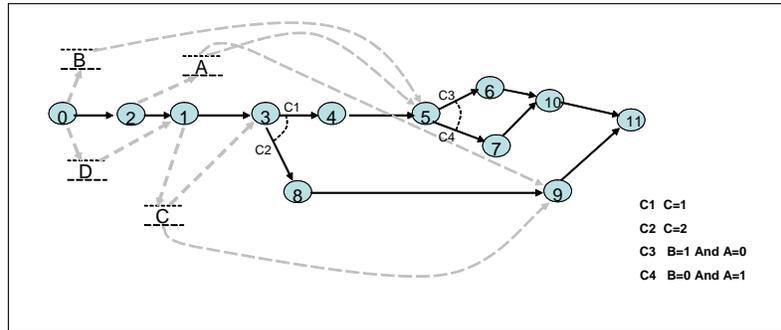33: **end for**
34: return(G)

Fig. 7. The resulting process after the propagation of activity a_1

propagated activity a_p, and if permitted (i.e. no dependencies), and the target activity is located after this block, a_p is propagated after the join activity of this block (e.g. skip the entire block). For illustration let us refer back to Figure 2, and have a look on the exception handler rollback section optimization. An exception handler with $a_{s1} = 8$, $a_{s2} = 1$,$a_t = 2$ will back-propagate activity 8 until reaching the Xor block $[3, 7]$. If all members of this block do not have any *tight* or *prec* relations with activity 8, it will skip the block and continue the back-propagation as before.

When the stop point a_{s2} is within the block boundaries, a_p will be handled according to the blocks property. Xor blocks execute only one path. Thus a_p should be duplicated to all paths within the block. In the path containing a_{s2}, a_p is propagated until reaching a_{s2}, or until reaching a dependent block member a_d. On the other paths a_p is propagated until reaching a_{s2}, or until reaching the join activity of a_{s2}. Taking for example Figure 6, in case that $a_{s2} = 5$ (selected alternative path is started with edge $e_{5,7}$, then activities 0, and 2 should be propagated starting with activity 2 (closer to a_{s2}). 2 is propagated until reaching the first Xor block. a_{s2} is within this block. Thus, 2 is propagated along the path $3 - 4 - 5$ until reaching 5 (since there is no dependent activity in between). On the other path $3-8-9-11$ the duplicated activity 2 is propagated until reaching activity 9 which has a dependency on activity 2. Unfortunately, activity 0 cannot be propagated since its successor is activity 1 which has a dependency on activity 0's output. The result of this scenario is presented in Figure 8.

In And blocks, all paths are executed. Thus, a_p will be propagated to the path that contains a_{s2} until reaching a dependent block member. If there is a dependent block member a_d on other paths within the block, a synchronization edge from a_p to a_d will be added. Referring again to Figure 6, and assuming that activity 3 performs as an And split, activity 2 is propagated along the path $3 - 4 - 5$ until reaching activity 5 (since there is no dependent activity in between). Nevertheless, on the other path $3 - 8 - 9 - 11$, there is activity 9 which is dependent on activity 2. Note that in the original process activity 2 was designed to be executed before activity 9. Due to the propagation of activity
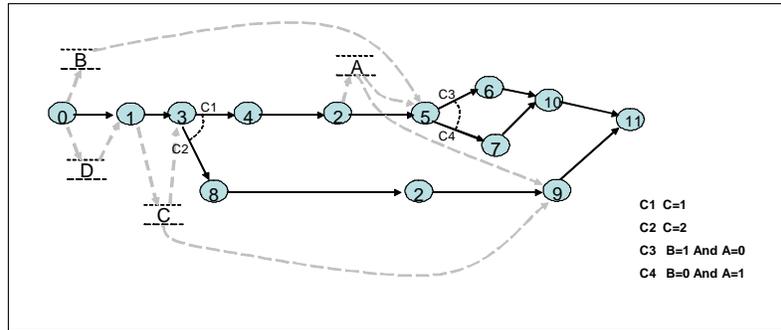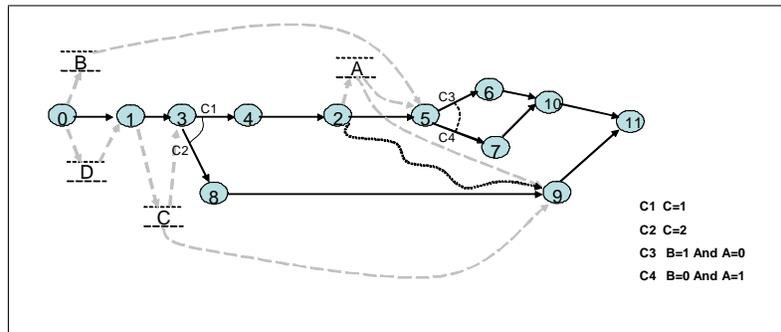
Fig. 8. Propagation into a xor block

Fig. 9. Propagation into an And block

2 into the And block, this sequentiality is lost. Thus, a synchronization edge should be added from activity 2 to activity 9. A *synchronization edge* in WSM nets guarantees that activities in parallel threads will be executed in a predefined order (in our case, activity 2 will be executed before activity 9 can start). The result of this scenario is presented in Figure 9.

5   Multi-exception Optimization

In this section we address the possible scenario where the process to be optimized is referred by multiple exception handlers. During design-time the designer may wish to optimize the process given several exception handlers $\{x_i, x_{i+1}, .., x_m\}$ for various activities, a prioritized list from the most important handler x_i. Once Algorithm 1 propagates the relevant activities for x_i, it is intended to run again for the following exception handler x_{i+1}. It may happen that the propagation of activities referring to x_{i+1} may interfere the minimal distance which was gained for x_i. In order to prevent this scenario, we tag all propagated activities after

each iteration with the *id* of its corresponding target activity during the propagation(Xor split node, or activity for re-execution, depending on the exception handler typeb). Thus, in the next propagation, already propagated activities a_k, are treated as blocks (starting with a_k and ending with a_x), such that activities either skip the entire block or remain before it. This way, previous propagations are not spoiled in terms of minimal distance with respect to later ones. This solution provide a heuristic optimization which may provide an non optimal solution, since a prioritized exception handler may restrict the optimization of many other less prioritized handlers.

For illustration let us return to the example in Figure 6. Suppose that the exception handler for activity 4 is the first one in the prioritized list, and this exception handler stops at activity 3. Activity 1 is propagated as closer as possible to activity 3, and is tagged with the *id* of activity 3, as resulted in Figure 7. Once referring to the next exception handler for activity 6, which stops at activity 5, activities 0, and 2 should be propagated. As mentioned in the previous section, activity 0 is denied from propagation, and only activity 2 is propagated. During this propagation once activity 2 get to be a predecessor of activity 1, it is verified that it could be propagated beyond activity 3 (i.e. the previous propagation block). once the verification passes activity 2 may skip the entire block (as demonstrated in Figure 8), or else remain before activity 1.

Finally, note that the mentioned algorithms in this section perform as an assisting tool for the designer. During design time it can provide the designer with **suggestions** for modifications to the given process model. Those modifications will be taken according to the designer's decision. This procedure may should become an iterative one, for allowing the designer to define the process in a more accurate way. Take for example the process in Figure 8, where activity 2 is recommended for propagation between activity 4 and activity 5. Receiving this recommendation, the designer may become aware that in this process, activity 2 must be executed before activity 4. Thus, a mandatory relationship of precedence should be defined between these two activities, and therefore the process should be re-analyzed.

6   Conclusion

In this paper, we propose a mechanism for design time optimization of exception handlers – in particular, by restructuring of the process graph. We show how changes in a process graph can minimize the costs related to exception handler execution. Our goal is to develop an approach that allows interactive (semi-) automatic exception handler optimization for arbitrarily complex business processes, balancing the difficulties faced by current workflow models and the control of a designer over the business process.

7   Acknowledgement

References

1. A. Agostini and G. De Michelis. Improving flexibility of workflow management systems. In W. van der Aalst and J. Oberweis, editors, *BPM: Models, Techniques, and Empirical Studies*, pages 218–234. Springer Verlag, 2000.
2. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 249–259, May 1987.
3. M. Golani. Flexible business process management using forward stepping and alternative paths, 2005. Ph.D. thesis, The Technion israel institute of technology.
4. M. Golani and A.Gal. Flexible business process management using forward stepping and alternative paths. In et al. W. van der Aalst, editor, *Lecture Notes on Computer Science, 3649*, pages 48–63. Springer Verlag, 2005. Proceedings of the Business Process Management International Conference, BPM 2005, Nancy, France, September 06-08, 2005.
5. C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Trans. Software Eng.*, 26(10):943–958, 2000.
6. M. Reichert and P. Dadam. Adept$_f$*lex*-supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems (JIIS)*, 10(2):93–129, March-April 1998.
7. S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.
8. W.M.P van der Aalst et al. Advance workflow patterns. In O. Etzion and P. Scheuermann, editors, *Cooperative Information Systems, 8th International Conference, CoopIS 2000, Eilat, Israel,Proceedings*, Lecture Notes in Computer Science 1901, pages 18–29. Springer, 2000.
9. Worflow management coalition. the workflow reference model (wfmc-tc-1003), 1995. http://www.wfmc.org.