

Flexible Business Process Management using Forward Stepping and Alternative Paths

Mati Golani, Avigdor Gal*
Technion - Israel Institute of Technology
{iemati,avigal}@ie.technion.ac.il

Abstract. The ability to continuously revise business practices is essential to organizations aiming at reducing their costs and increasing their revenues. Rapid and continuous changes to business processes result in less control over the executed activities. As a result, the ability of process designers to produce solid, well-validated workflow models is limited. Workflow management systems (WfMSs), serving as the main vehicle of business process execution, should recognize these risks and become more dynamic to allow the required business flexibility. In this paper, we propose a dynamic mechanism that allows backtracking and forward stepping at an instance level. This mechanism analyzes the feasibility of applying certain modifications to running instances and provides an efficient algorithm that avoids redundant operation activation. We believe that this mechanism can bolster the ability of a business process management system to deal with unexpected situations and to resolve, in runtime, scenarios in which such resolution both is called for and does not violate any business process constraints. Throughout this paper, we use the paradigm of Web services to demonstrate the capabilities of the proposed mechanism.

1 Introduction

Rapidly changing business environments require organizations to continuously revise their business practices, seeking better business opportunities and continuously aiming at reducing their costs and increasing their revenues. In the last decade, businesses have turned to technological solutions to assist them in this task. The use of electronic means to commerce, data mining, and customer profiling are all recent technological developments that penetrate business activities. One of the most recent technological developments is the use of Web services, components with a well-defined interface that are embedded in cross-organizational business processes. Using Web services, the functional aspects of business applications are encapsulated [19], with interfaces defined using standards such as BPEL4WS [3], and invocation controlled using approaches such as Service Oriented Architecture (SOA). Web services promise to deliver greater choice and flexibility to business processes.

* The work of Gal was partially supported by two European Commission 6th Framework IST projects, QUALEG and TerreGov, and the Fund for the Promotion of Research at the Technion.

Rapid and continuous changes to business processes carry with it risks, due to shorter (or even nonexistent) design time and less control over the executed activities. As a result, the ability of process designers to produce solid, well-validated workflow models is limited. Workflow management systems (WfMSs), serving as the main vehicle of business process execution, should recognize these risks and become more dynamic to allow the required business flexibility. To illustrate this point, we next present two examples, involving Web services. First, we observe that the development of Web services is an ongoing task, and new and improved services are continuously replacing existing ones. Currently, WfMSs provide little support to the reexecution of successfully processed tasks for running instances, even if the gains from such reexecution outweigh the costs. As another example, observe that Web services merely provide syntactic information regarding their input, output and processing logic, through standards such as WSDL [20]. In most cases, such descriptions fail to convey all necessary constraints and restrictions. Modeling using Web services, therefore, is likely to make the validation of workflow models more difficult [8], and more exceptions at run-time are to be expected. Efficient exception handling is a fundamental component of WfMSs and is critical to their successful implementation in real-world scenarios [1].

The motivation for this work, as illustrated above, is the need for flexible and dynamic WfMSs to support the growing number of exceptions that cannot be designed a priori, due to poor design or the lack of sufficient information regarding the internal logic of Web services. In this paper, we propose a dynamic workflow mechanism that allows backtracking and forward stepping at an instance level. This mechanism works by analyzing the feasibility of applying certain modifications to running instances and implementing an efficient algorithm that avoids redundant operation activation as a result of instance modification. In particular, the introduction of a new Web service may trigger backtracking of an instance to an activity in which the new Web service is performed, and then forward stepping while utilizing, to the extent possible, previously executed activities. In the case of an exception, the proposed algorithm identifies a feasible alternative that avoids the failing activity or communication channel. In this scenario, again, we backtrack to an activity from which it is considered safe to step forward.

Our approach is based on and extends the approach described in [7]. Therefore, we provide a rollback mechanism followed by a forward execution. We extend the existing proposal by providing a precise location in the workflow graph for the rollback target and a semi-automated forward execution (involving relevant agents when needed). We believe that this mechanism can bolster the ability of a business process management system to deal with unexpected situations and to resolve, in runtime, scenarios in which such resolution is both called for and does not violate any business process constraints.

We base our workflow model on ADEPT WSM nets [13]. It combines a graphical representation with a solid formal foundation, taken from graph theory, allowing both reasoning and an execution engine [14].

Our specific contributions are as follows:

- We provide an analysis of a workflow model, based on WSM nets, that generates a conceptual framework in which backtracking and forward stepping can be evaluated and implemented.
- We provide efficient algorithms for alternative route identification (at design time or run time) and forward stepping (at run time), to allow dynamic modifications to workflows.
- We introduce the concept of a meta-process, an efficient and a fully automatic mechanism (at the WfMS level) for activating the proposed algorithms.

1.1 Related work

Exception and modification handling – *i.e.*, the way a workflow system responds once an exception or a modification notification occurs – has been discussed in the literature for some time. The system may react in such cases either by terminating a process or handling an exception [10, 4]. Sadiq et al. in [15] classified the latter option as one of the available modification policies (which was called *Adapt*) to a given change in a running process. This change is due to an unexpected exception, so the process should be handled differently than originally designed. Yet the authors do not define how to infer this modification.

Generally speaking, exception handling involves compensation flows [5]. Compensation flows provide rollback, a set of undo actions. These flows are predefined. If compensation does not exist, the workflow operator may be willing to accept inconsistencies in which a completed activity is not voided. For example, assume an activity provides a customer with bonus points, on the assumption that a purchase will be made. Then another activity is chosen, which also awards bonus points. For a successful termination of the workflow (*e.g.*, a sale), an operator may be willing to grant double bonus points in this case.

Eder et al. describe several types of compensation in [6], and provide a three-step mechanism to handle exceptions [6, workflow recovery] [7]. The first step entails rollback based on compensation type of activities in the workflow graph. In the next step, an agent determines whether to continue backward or to take an alternative path. The final step is a forward execution (which could lead to the same point of failure). In the event of rollback, existing work [7] does not specify the stop point, implying that this point represent the decision on whether to continue. However, in many cases the parameter which drives this decision has been set before this point. Furthermore, these mechanisms are static (*e.g.*, during build time) [5, 12, 6]. Our approach detects the actual/optimal stop point (via analysis), and can provide in run time an alternative execution that overtakes the failed activity.

A dynamic approach was presented by Hwang et al. [11]. Here, a failure recovery language supports multiple exceptions per activity, and applies rollback using ECP (end compensation point). This language uses the process parameters in order to determine its flow. The drawback of this approach is that it fails to make use of the user’s insight (and output). It is impossible to accurately forecast all user intentions, and under different circumstances, individual users may make different choices based on the same input. Thus, the user’s output is essential.

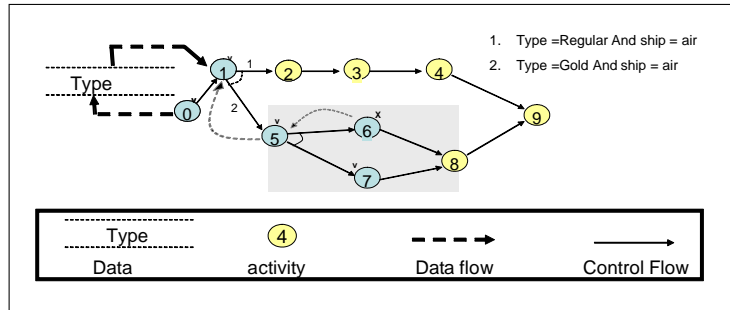


Fig. 1. An example of a business process

The rest of the paper is organized as follows: Section 2 introduces a motivating example. In Section 3 we present the workflow graph-based model. In Sections 4-6 we present forward stepping, alternative path, and parametric analysis mechanisms, respectively. Finally, in Section 7, we introduce the architecture for implementation, and show the use of meta-processes.

2 Illustrative Example

As an example, consider a process that handles registrations for package tours (see Figure 1). This process uses some local applications (member deals), as well as Web services (hotel registration: activity 0, flight reservation: activity 1) with some special offers available to gold members only (activities 5-8). We use two examples to illustrate the needs of a flexible business process. The first involves introduction of a new Web service that offers better hotels for cheaper prices, at a time when the process instance is already handling membership registration (activity 4), and hotel registration and flight ticket activities are already completed. The second event involves a failure of the special offer system for a gold customer (activity 6).

In the first example, nothing has gone wrong, but the “world” has changed (a new service has become available). Taking advantage of this new service may affect the customer’s total cost, given penalties for canceling an existing order and the cost of creating a new one. Other activities or services may also need to be compensated or reexecuted as a result of this update. For instance, activity 1 may require compensation if the original flight dates were modified based on the new hotel reservations. These costs must be quantified before the customer decides whether to continue with the original plan or to use the newly available service.

In the second event, the system will benefit by using a different path that allows successful completion of the business process while bypassing the special offers. Consider the example depicted in Figure 1. Activities 0, 1, 5, and 7 were performed in this instantiation, yet a failure at activity 6 blocks the process and prevents its completion. Alternative paths to the current path, to be formally

defined in Section 3, are those paths in the graph that possibly lead to a successful termination of the business process, yet do not contain activity 6. Since activity 1 leads to activities 2 and 5 using a Xor condition, a rollback procedure to activity 1 would enable use of an alternative path to 9 through activities 2, 3, and 4.

In this paper we address only processes in which the alternative path has a real semantic alternative meaning. That means that an executed instance along one path can be logically executed (albeit, at a possibly higher cost) along the alternative path as well (as in the Regular/Gold customer example).

3 Workflow Model

In this section we define basic constructs in workflow graphs, to be used later in the paper. The classification of workflow constructs is not new and has been discussed in various works (*e.g.*, [16]).

A workflow model can be described as a graph (ADEPT WSM net) $G(V, E)$ ($V = (V_a \cup V_d)$; $E = (E_c \cup E_d)$), where V_a is a set of activities, V_d is a set of data parameters, E_c is a set of control edges, and E_d is a set of data edges. For simplicity, whenever possible, we will refer to the reduced graph $G' = G(V_a, E_c)$. Data flows (as appear in Figure 1) are discussed in Section 6.

An activity a in V_a has *in-degree*(G', a) incoming edges and *out-degree*(G', a) outgoing edges. Whenever it becomes clear from the context, we eliminate the graph reference and refer to *in-degree*(a) and *out-degree*(a). A *path* in G is a set of activities such that any two consecutive activities on the path are connected by an edge in E_c . We denote a path from a_i to a_j by (a_i, \dots, a_j) . The *length* of a *path*(a_i, \dots, a_j) (denoted *length*(a_i, \dots, a_j)) is the number of edges in (a_i, \dots, a_j) . Finally, *Minlength*(a_i, a_j) is the length of the shortest path in G that starts at a_i and ends at a_j .

We next define two graph constructs, namely splits and joins, based on the Workflow Management Coalition standard [17]. A Xor *split* a is a node (activity) with multiple outgoing edges (*out-degree*(a)>1), only one of which can be followed in the execution flow. The decision as to which edge to follow is based on the satisfaction of mutually exclusive conditions that are typically associated with the outgoing edges. Let $c_{a,a'}$ be a DNF (Disjunctive Normal Form) Boolean statement with a set of variables $Var(c_{a,a'})$ that must be satisfied in order to pass from activity a to activity a' . Activity 1 in Figure 1 is an example of a Xor split. Each Xor split a is associated with a Xor join (e.g., activity 9 in Figure 1), an activity common to all paths that start from a . During runtime, when reaching a , the workflow engine evaluates the conditions on each of a 's outgoing edges, and continues the execution along the edge whose associated condition is satisfied. The Xor join activity acts as a synchronization point in the execution.

An *And split* is a node with multiple outgoing edges whose execution flow follows all outgoing edges by parallel threading. Activity 5 in Figure 1 is an example of an And split. Threads of an And split a need to be synchronized at an *And join*, which is also a node in the graph that is common to all paths that start from a . Activity 8 is an example of an And join for activity 5.

Definition 1. Xor split point Let $G^c = (V, E)$ be a workflow graph, and a be an activity in V . A Xor split point of a is a Xor split a_i with a Xor join a_j such that a_i is a predecessor of a and a_j is a successor of a .

Definition 2. NXSP Nearest Xor split point of a ($NXSP(a)$), is a Xor split point of a , a_i , which satisfies that any other Xor split point of a (a_j) is also a Xor split point of a_i .

And split point and NASP are similarly defined. Using the basic definitions given above, we now define blocks in a graph. Let $G^c = (V, E)$ be a workflow graph and let a_i be a Xor split and a_j be the Xor join associated with a_i . A Xor block of a_i is a subgraph of G^c induced by the nodes of all paths (a_i, \dots, a_j) in G^c . Similarly, given an And split a_i and the associated And join of a_i, a_j , an And block of a_i is a subgraph of G^c induced by the nodes of all paths (a_i, \dots, a_j) in G^c . For example, the induced subgraph of activities $\{5, 6, 7, 8\}$ in Figure 1 (marked with grey rectangle) is an And Block. It models two threads that start after the execution of activity 5 and synchronize before the execution of activity 8.

Clearly, any activity a is within a Xor block defined by its Xor split point (can be null) and its associated Xor join. In particular, a is within a Xor block defined by $NXSP(a)$ and its associated Xor join.

Definition 3. Alternative paths Let $G^c = (V, E)$ be a workflow graph with a sink f , and let $P_1 = (a_i, \dots, a_j)$ and $P_2 = (a_i, \dots, a_k)$ be paths in G^c . P_1 is an alternative to P_2 (and vice versa) if the following four conditions hold:

1. a_i is of type Xor Split.
2. There is no activity a in $V \setminus \{a_i\}$ such that a is in P_1 and a is also in P_2 .
3. Any path (a_j, \dots, f) in G^c does not include an activity in P_2 .
4. Any path (a_k, \dots, f) in G^c does not include an activity in P_1 .

It is worth noting that P_1 and P_2 share a common initial activity a_i . As an example, consider the alternative paths (1, 2, 3, 4) and (1, 5, 6, 7, 8) in Figure 1. Note that the paths (5, 6) and (5, 7) are **not** alternative paths, since activity 5 is not of type Xor Split (both activity 6 and activity 7 are part of the same And block).

The importance of Xor blocks in our analysis is related to the ability to provide an alternative paths analysis. In Figure 1, the Xor Block includes the entire graph save activity 0, and thus an alternative path for any activity (excluding activity 9) will start from activity 1. Therefore, once activity 6 fails, the Xor Block to which activity 6 belongs allows an alternative execution, using the paths that contains activities $\{1, 2, 3, 4\}$. We will present an algorithm for identifying alternative paths in Section 5.

We next discuss the normalization of Xor and And blocks. A normalized (Xor or And) block is a block in which neither the outgoing edges of the split activity, nor the incoming edges of the join activity, are connected to any activities outside the block. This property matches the WFMC definition (in interface 1) of *full-blocked workflows*. Formally,

Algorithm 1 Forward stepping

Input: $G(V_a \cup V_d, E_c \cup E_d)$, a_i -first activity path, a_x - the stop activity (optional)

Output: *potentialList* - a list of potential activities to be reexecuted, *semanticList* - a list of activities to be semantically executed

add $a_i.successors$ into Q // Q is a Queue

put a_i into *visitedList*.

put $a_i.outputParameters$ into D .

while Q not empty **do**

 put $a_k = dequeue(Q)$ into *visitedList*

if $\exists a_k.inputParameter \in D$ **then**

 add a_k to *potentialList*

 add $a_k.outputParameter$ to D

else

 add a_k to *semanticList*

end if

if $a_k \neq a_x$ **then**

 add a_k 's executed successor activities (as appear in $Inst(G)$) to Q . Add only activities that satisfy $predecessor(a) \subset visited$

end if

end while

return *potentialList*, *semanticList*

The forward stepping algorithm is given in Algorithm 1. Looking at the process structure, this algorithm - given it doesn't use instance data - can be invoked asynchronously with runtime instances (*e.g.*, in advance). *PotentialList* holds potential activities (derived from the process structure) for reexecution, of which only those satisfying the *Reexecute* condition in Eq 1 should be re-executed. During runtime, some of these activities may receive the same input values as in the original execution. Therefore, despite their dependency on other reexecuted activities, we expect their previous output to be valid (due to the validity property). In such cases, semantic execution is sufficient, and there is no overhead cost for reexecution.

5 Alternative Paths Detection

In the case of an exception, undefined in advance, the workflow engine should rollback to an activity in the graph from which it can provide an alternative path to complete execution of the business process. We will refer to this activity as a *rollback point*. In an extended version of this work, we will elaborate on the heuristics of finding the best rollback point, and design time considerations in determining the suitability of alternative paths. This section details the necessary steps for rollback.

Definition 5. Rollback point: Let a_i be an activity in a normalized workflow graph $G' = (V, E)$ with a sink f . A rollback point of a_i in a given instance $Inst(G')$ is an activity a_j that satisfies the following conditions:

1. a_j was activated during $Inst(G')$ (i.e., a_j 's state in $Inst(G')$ is "com-pleted").
2. There is a path $P_1 = (a_j, \dots, a_i)$ in $Inst(G')$ of which all activities in $P_1 \setminus a_i$ are in state "completed".
3. There is a path $P_2 = (a_j, \dots, f)$ in G' , such that P_2 is an alternative path to P_1 .

A nearest rollback point of a_i in a given instance $Inst(G')$ of G' is a rollback activity a_k such that $minlength(a_k, a_i) \leq minlength(a_l, a_i)$ for any rollback point a_l of a_i .

Theorem 1. : Let a_i be an activity in a workflow graph $G' = (V, E)$. The nearest rollback point of a_i is $NXSP(a_i)$.

We refrain from presenting the proof of Theorem 1 in this paper due to space considerations.

Rollback can be classified into three types, namely *single threaded*, *parallel threaded*, and *hybrid*. We will define each of these types and specify the rollback activities needed for each type, using Theorem 1 as a guideline.

Single-threaded rollback is a rollback in which the failing activity falls within a single thread. This means that upon failure, the rollback procedure should be applied only to this thread. The following rollback activity should be taken in a single-threaded type:

$$a_j = NXSP(a_i). \text{ Rollback until reaching } a_j$$

Parallel-threaded rollback refers to a rollback in which the failing activity falls in one of multiple running threads. That means that there is an And split ($NASP(a_i)$) in the path($NXSP(a_i), a_i$). In this case, the rollback is performed for all parallel threads within the same And block, until the And split activity of the block containing the failing activity (marked as B_a) is reached. At this point it continues as single-threaded until reaching the nearest Xor split point. In the example given in Figure 1, there are two parallel threads running when activity 6 fails. The other thread, which executes activity 7, is forced to rollback until reaching activity 5, at which point the process continues as single-threaded. The following rollback activity should be taken in a parallel-threaded type:

Rollback all current executing and completed activities
within B_a and proceed rollback as single-threaded.

Hybrid rollback is a rollback in which the failing activity a in $Inst(G')$ is part of a single thread, but activities in $Inst(G')$ are part of an And block prior to the execution of a . For example, in Figure 1 assume that activity 8, which runs as single-threaded, fails. Since the process contains an And-block (activities 6 and 7 running in parallel), the rollback mechanism should apply to the entire And block and continue with the rollback until reaching $NXSP(8) = 1$. The

following rollback activity should be taken in a hybrid-threaded type:

$a_j = NXSP(a_i)$. Rollback until reaching a_j . For each
And-block, rollback all activities in the block and continue.

Algorithm 2 summarizes the mechanism for rollback discussed above. The correctness of Algorithm 2 stems immediately from Theorem 1. It is worth noting that single-threaded rollback is a special case of hybrid-threaded rollback, and therefore the algorithm refers only to the latter.

Algorithm 2 Rollback

- 1: **Input:** $G, Inst(G)$ -Instantiation, a_i -activity from which the rollback starts.
 - 2: **Output:** $Inst'(G)$ - revised instantiation. Rollback of activities is performed to a_i 's nearest rollback point.
 - 3: Process:
 - 4: On the failure of activity a_i , $a_A = NASP(a_i)$ and $a_X = NXSP(a_i)$, if exist.
 - 5: **if** $a_A = null$ **then**
 - 6: Rollback as Hybrid threaded.
 - 7: **else if** $length(a_X, a_i) < length(a_A, a_i)$ **then**
 - 8: Rollback as Hybrid threaded.
 - 9: **else**
 - 10: Rollback as parallel threaded.
 - 11: **end if**
-

In case of a failure in one of the threads of an And block (e.g., activity 6), one needs to rollback other threads as well (e.g., activity 7 in Figure 1). However, there can be scenarios in which there is no need for rollback of the concurrent threads. In particular, if the failing activity occurs in a Xor block within an And block, an alternative path that does not require the rollback of all of the And block activities can be provided. This case is handled in Line 8 of the algorithm.

$NASP$ and $NXSP$ can be pre-assigned by analyzing the graph at design time. At each rollback step the compensation activity is assumed to execute in $O(1)$ (a more refined approach which addresses more complicated executions is deferred to an extended version of this work). There are $minlength(NXSP(a_i), a_i)$ steps to be taken, which is bounded by the cardinality of E . Therefore, the algorithm complexity is $O(E)$.

6 Parametric Modification Analysis

This section discusses scenarios, such as exception handling, that are handled with alternative paths. Once an alternative path has been discovered (see Section 5), it is necessary to evaluate the pre-conditions for performing this new path, and to request a change of values to satisfy these pre-conditions. We therefore turn our attention to the data flow of a business process. As an example, consider once more Figure 1, which introduces a data flow of a single data item, $Type$. This data item is updated during the execution of activity 0, and is retrieved by

activity 1. Using common notation [14], the data flow is marked using dashed double-line arrows. In what follows, we denote by $Update(var, a)$ the nearest predecessor of a in which the variable var has been updated. This information can be generated offline and kept with each node, so that accessing it can be done in $O(1)$.

The parametric modification analysis is performed in two steps. The first entails identifying a set of variables whose modification would allow the use of an alternative path. The next is to identify the agents that have assigned the original values to these variables, and to request a change that would allow the use of the alternative path. We here detail each of these steps.

6.1 Satisfying Changes

Going back to Figure 1, recall that the original path to be taken was the path (0, 1, 5, 6, 7, 8, 9). Once activity 1 has been performed, the decision on whether to continue to activity 2 or to activity 5 is based on a mutually exclusive condition (regular or gold customer). Therefore, it becomes evident that the condition that enables us to proceed to activity 2 cannot be satisfied unless some of the variables are assigned different values.

For a given instance $Inst(G')$, each variable var in $Var(c_{a,a'})$ is assigned a value. Let $D(c_{a,a'}, Inst(G'))$ be a **set of sets** of assignments of the type $var = val$ from $Inst(G')$, for which $c_{a,a'}$ cannot be satisfied. In the example given in Figure 1, $Var(c_{1,2}) = \{Type, Shipping\}$, and $D(c_{1,2}, Inst(G')) = \{Type = \text{“Regular”}\}$, since under this instance, $Type = \text{“Gold”}$.

Given an instance $Inst(G')$ with an assignment $var = val$, where var is in $Var(c_{a,a'})$, one may consider a modified instance $Inst'(G') = Inst(G') \setminus \{var = val\} \cup \{var = val'\}$ in which $var = val$ is replaced with $var = val'$. For example, a modified instance may include $Type = \text{“Regular”}$ instead of $Type = \text{“Gold”}$.

Definition 6. *minimal satisfying change:* Let $c_{u,v}$ and $Inst(G')$ be defined as before and let

$D(c_{a,a'}, Inst(G')) = \{set1\{var_{11} = val_{11}, \dots, var_{1n} = val_{1n}\}, set2\{var_{21} = val_{21}, \dots, var_{2m} = val_{2m}\}, \dots\}$. Note that each set may be in different length, and some sets may share the same variables. A *satisfying change* to $Inst(G')$ is a set of assignments $\{var_{i1} = val'_{i1}, \dots, var_{in} = val'_{in}\}$ such that $c_{a,a'}$ can be satisfied under

$$Inst'(G') = Inst(G') \setminus \{var_{i1} = val_{i1}, \dots, var_{in} = val_{in}\} \cup \{var_{i1} = val'_{i1}, \dots, var_{in} = val'_{in}\}$$

A *minimal satisfying change* is a satisfying change such that L (Eq 2) is the minimal of all possible satisfying changes. The *max* function is required since a *DNF* expression contains sets of predicates. Each set contain simple predicates with an *And* relation between them. All predicates in this set have to be satisfied in order to satisfy the set.

$$L = \max_{i=1}^n \left(\minlength \left(Update \left(var_i, a \right), a \right) \right) \quad (2)$$

Definition 6 defines a minimal change to be the set of assignments in $Inst(G')$ that can satisfy $c_{a,a'}$. Consider, for example, $c_{1,2}$ that includes the following statement:

$$(Type = \text{“Regular”} \wedge Shipping = \text{“air”}) \vee (Destination = 972 \wedge City = \text{“TLV”})$$

and assume that all variables but $Type$ are updated before activity 0. In this case, $Var(C_{1,2}) = \{Type, Shipping, Destination, City\}$. Assume an instance in which $Type = \text{“Gold”}$, $Shipping = \text{“air”}$, $Destination = 33$, and $City = \text{“Nancy”}$. Therefore, $D(c_{1,2}, Inst(G')) = \{\{Type = \text{“Regular”}\}, \{Destination = 972, City = \text{“TLV”}\}\}$. The *minimal satisfying change* would be $\{Type = \text{“Regular”}\}$.

It is worth noting that Definition 6 minimizes the maximal number of activities for which rollback is needed. Such a definition seems reasonable when the rollback of any activity has the same cost, from the user’s point of view. A more general approach would require the definition of a cost model to evaluate the impact of an activity rollback as well as a variable change, and to minimize this impact. We defer the introduction of this approach to the extended version of this work.

6.2 Variable Modification

Once the minimal satisfying change is computed, we can identify the variables that need to be modified. From the workflow model, using either a-priori information or mining procedures [9, 2, 18], one can identify the activity where those variables are modified. For example, in Figure 1 an exception occurs while executing activity 6. $NXSP$ is detected as activity 1 and the minimal satisfying change is $\{Type = \text{“Regular”}\}$. This change can be set in activity 0. Let a_i be an activity in which a variable change is required. There are two possible sources for updated values, as follows:

A user-defined value: This is a value inserted by the agent that executed a_i . In this case, the user will be presented with a request for reexecution of the activity, with the specific condition needed to allow execution of the alternative path.

A derived value: This is an expression whose input includes both data flow and user input. If the user input affects the data value in such a way that the desired data value is feasible, the user’s approval is requested for reexecution of a_i with a specific range of valid input to allow the alternative path execution.

Upon approval, the business process rolls back to a_X (see Section 5), and forward stepping is performed from a_X to $NXSP(a_i)$. Once $NXSP(a_i)$ is completed, the expression is evaluated again, but this time the evaluation result redirects the execution to the alternative path.

In the case the modifying change request is rejected, the next minimal satisfying change is checked, and so on, until all changes have been exhausted. Then, the second nearest rollback point is computed, and the same procedure is applied to it. The *second nearest rollback point* (based on Definition 5), can be computed recursively as the nearest rollback point of $NXSP(a_i)$. Thus, the

Algorithm 3 Parametric modification algorithm

```
1: Input: Graph  $G'$ ,  $Inst(G)$  -Instantiation,  $a_i$  -failed activity.
2: Output:  $parameters$  - The parameters to be modified after Role approval.
3: Process:
4: repeat
5:   //execute over the nested Xor blockes
6:    $A1 = NXSP(a_i)$ .
7:   for each  $e_{j,k}$  (an outgoing edge from A1) that does not lead to  $a_i$  do
8:     get  $C_{j,k}$ 
9:     get  $Var(C_{j,k})$ 
10:  end for
11:   $L = getD(c, Inst(G))$  //list of satisfying changes.
12:  Sort  $L$  by increasing length, using equation 2.
13:  for  $i = 0$  to  $L$  size do
14:     $parameters[var, val'] = L[i]$ 
15:    for all set of assignments do
16:       $var = val'$ 
17:    end for
18:    if all assignments accepted then
19:      return ( $parameters$ )
20:    else
21:      continue // to the next satisfying change.
22:    end if
23:  end for
24:   $A1 = NXSP(A1)$  // next Xor split point
25: until  $A1 = null$ 
26: return null
```

algorithm will recursively compute the same actions over the next nested Xor block (*i.e.*, $NXSP(NXSP(a_i))$). This process is summarized in Algorithm 3.

In the worst case the algorithm will iterate over all Xor split points, scanning all graph edges ($O(E)$). For each edge we generate (line 11) a list L of cardinality $|L|$, sorted (line 11) in $O(|L|\log|L|)$. Therefore, the total complexity of Algorithm 3 is $O(E|L|\log|L|)$. It is worth noting that L may be exponential in $Var(C)$. However, under a reasonable assumption of rather simple conditions with a small $Var(c)$ with a constant upper limit, the algorithm complexity is $O(E)$.

6.3 Forward stepping - revisited

Given that a change to activity a_i was approved by the relevant agent, at least one of the output parameters (P_i) of an activity a_i must have been modified. The naïve approach assumes that the process can move forward in a semantic manner (see Section 4) until reaching the relevant Xor split point (a_x). However, along the path (a_i, a_x) there are activities which may be affected by the modified value of one of the output parameters of a_i . An activity which uses a modified parameter value as input should not be executed semantically, since the output parameters values may have been revised based on the modified input data. The mechanism for such an approach has been discussed in Algorithm 1 (combined

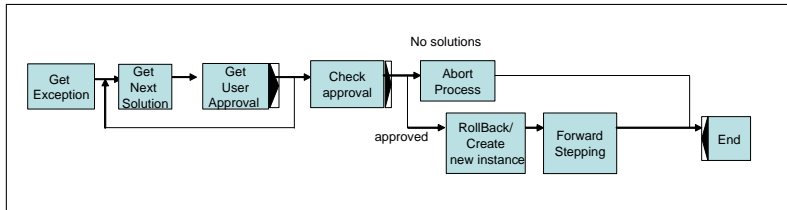


Fig. 2. The Meta-process description for exception handling.

with Eq 1). In this case, the algorithm is executed with a known stop activity a_x .

7 Architectural Considerations

We used the core functionality of a WfMS system to orchestrate our solution. A meta process is a process that manages other processes. Figure 2 presents a description of a *meta process* for exception handling. Slight modifications are needed to generalize it to the more general case. In our case, once a problem/opportunity is monitored, the meta process is invoked and its activities are executed to provide the best solution. Each solution should be confirmed by the relevant agent prior to the semi-automated execution of the underlying process. Upon approval, those activities that are not affected by user input are semantically executed (assuming validity), while other activities are referred to their original responsible agent for execution. The result is a single meta process that can interact with all running processes using the system infrastructure and constructs, and that provides a transparent mechanism to handle such ad-hoc changes via backtracking and forward stepping.

A meta process invokes a monitor that acts as a special workflow client (see Figure 3(a)). The monitor receives modification notifications and exception-oriented messages (*e.g.*, work items), and in response creates an instance of a process that requests a parameter change from the relevant agent. If the reply (again as work item) is negative, then the monitor seeks the next available solution and makes another request to an appropriate agent. This continues iteratively until there are no more solutions to suggest (as discussed in Algorithm 3). Once a positive answer arrives at the monitor, it rolls back to the required activity a_y (or creates a new instance that imitates and semantically executes the original instance activities until reaching a_y), and then starts the reexecution and semantic execution of the preceding activities, until reaching an activity that was not on the original path.

A prototype was built over the ADEPT workflow system (see Figure 3(b)). The BP monitor reads messages from the work items list. An exception is stored in the *exception Store*, while the analyzer analyzes the process graph and creates a list of solutions (for this exception) sorted from best to worst according to an

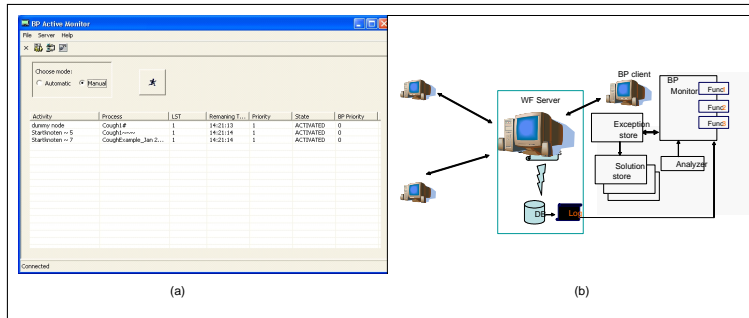


Fig. 3. (a) Business process monitor prototype (b) Architecture

estimation function. This list is stored in the *solution store*. At each iteration the *exception store* requests a new solution from the *solution store*.

8 Conclusion

In this paper, we propose a mechanism for efficient management of flexible business processes – in particular, for forward stepping and backtracking. For illustration purposes, we demonstrate our techniques with two somewhat different scenarios. First, we show how changes in Web services can be dynamically embedded into a workflow model, minimizing the costs related to reexecution of previously performed activities. Second, we propose an improved exception handler using forward stepping, backtracking, and alternative paths. Alternative paths, while not the only possible exception handling tool, can serve in a broad variety of cases, and can be easily produced automatically, either in design time (serving as a recommendation) or at run-time (serving as a crisis management tool, in the absence of immediate valid solutions). Our goal is to develop an approach that allows (semi-) automatic dynamic management for arbitrarily complex business processes, balancing the difficulties faced by current workflow models and the control of a designer over the business process.

Future work involves a thorough analysis of utility in the context of reexecution and compensation of activities. Another intriguing direction is data integration for required services, since a replacement service may require data not needed by the original service.

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. R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In O. Etzion and P. Scheuermann, editors, *Advances in Database Technology - EDBT'98, 6th international Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, Lecture Notes in Computer Science 1337, pages 469–483. Springer, 1998.

3. Specification: Business process execution language for web services version 1.1. <http://www-128.ibm.com/developerworks/library/ws-bpel/>.
4. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999.
5. W. Du, J. Davis, and M.C. Shan. Flexible specification of workflow compensation scopes. In *GROUP*, pages 309–316. ACM, 1997.
6. J. Eder and W. Liebhart. Workflow recovery. In *CoopIS*, pages 124–134, 1996.
7. J. Eder and W. Liebhart. Contributions to exception handling in workflow management. In O. Burkes, J. Eder, and S. Salza, editors, *Proceedings of the Sixth International Conference on Extending Database Technology, Valencia, Spain, March 1998*, pages 3–10, 1998.
8. W. Gaaloul, S. Bhiri, and C. Godart. Discovering workflow transactional behavior from event-based log. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 3–18. Springer, October 2004.
9. M. Golani and S.S. Pinter. Generating a process model from a process audit log. In M. Weske W. van der Aalst, A. ter Hofstede, editor, *Lecture Notes on Computer Science, 2678*, pages 136–151. Springer Verlag, 2003. Proceedings of the Business Process Management International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003.
10. C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Trans. Software Eng.*, 26(10):943–958, 2000.
11. G.H. Hwang, Y.C. Lee, and B.Y. Wu. A new language to support flexible failure recovery for workflow management systems. In Jesús Favela and Dominique Decouchant, editors, *CRIWG*, volume 2806 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2003.
12. M. Kamath and K. Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *ICDE*, pages 334–341. IEEE Computer Society, 1998.
13. M. Reichert and P. Dadam. Adept_{flex}-supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems (JIIS)*, 10(2):93–129, March-April 1998.
14. S. Rinderle, M. Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.
15. S. Sadiq, O. Marjanovic, and M. E. Orłowska. Managing Change and Time in Dynamic Workflow Processes. *International Journal of Cooperative Information Systems*, 9(1-2):93–116, 2000.
16. 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.
17. Workflow management coalition. the workflow reference model (wfmc-tc-1003), 1995.
18. workflow management coalition 1998. interface 5 - audit data specification. technical report wfmc-tc-1015 issue 1.1. workflow management coalition.
19. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M ter Hostede. Analysis of web services composition languages: The case of bpel4ws. In Song et al., editor, *Conceptual Modeling - ER 2003 - 22nd international Conference on Conceptual Modeling, Chicago, IL USA, October, 2003, Proceedings*, Lecture Notes in Computer Science 2813, pages 200–215. Springer, 2003.
20. Specification: Web services description language (wsdl) version 2.0. <http://www.w3.org/TR/wsdl>.