Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL

Rania Khalaf[1], Dieter Roller[2], and Frank Leymann[2]

[1] IBM TJ Watson Research Center, 1 Rogers St, Cambridge MA 02142, USA
rhkalaf@us.ibm.com
[2] Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
{Dieter.H.Roller,Frank.Leymann}@iaas.uni-stuttgart.de

Abstract. When automating work, it is often desirable to compensate completed work by undoing the work done by one or more activities. In the context of workflow, where compensation actions are defined on nested 'scopes' that group activities, this requires a model of nested compensation–based transactions. The model must enable the automatic determination of compensation order by considering not only the nesting of scopes but also the control dependencies between them. The current standard for Web services workflows, Business Process Execution Language for Web Services (WS-BPEL), has such compensation capabilities. In this paper, we show that the current mechanism in WS-BPEL shows compensation processing anomalies, such as neglecting control link dependencies between nested non-isolated scopes. We then propose an alternate approach that through elimination of default handlers as well as the complete elimination of termination handlers not only removes those anomalies but also relaxes current WS-BPEL restrictions on control links. The result is a new and deterministic model for handling default compensation for scopes in structures where: (1)both fault handling and compensation handling are present and (2)the relationships between scopes include both structured nesting and graph–based links.

Keywords: WS-BPEL, Compensation, Transactions, Error handling, Workflow.

1 Introduction

WS-BPEL [18] is the business process modeling language for Web services. The language is rich in functionality, providing capabilities for fault and compensation handling. Fault handling provides a process fragment that runs if a fault is raised by any of a set of running activities. Compensation handing provides a process fragment that runs to reverse the cumulative work of a set of successfully completed activities. Compensation is usually run due to a fault in another part of the process. A set of WS-BPEL activities is defined by grouping them in a 'scope'. Scopes in WS-BPEL are also used to provide additional capabilities such as scoping of variables, event handlers that can run multiple times as long as the scope is active, and termination handlers that are run if the scope is running when its parent scope is trying to exit. For fault, compensation, and termination handlers, WS-BPEL provides default behavior for each that is attached

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 286–303, 2009.

on every single scope that does not explicitly provide one of them. This behavior is complicated by WS-BPEL's mixed control model which allows control dependencies to be defined using both/either structured activities and explicit control links. Structured activities like 'sequence' or 'flow' impose control logic on activities nested within them, while a control link from a source activity to a target activity specifies that the target activity must not start until the source activity has completed. WS-BPEL allows control links to cross the boundaries of structured activities, for example going from an activity inside one 'flow' to another activity nested inside another 'flow'. In this paper, we argue that WS-BPEL's default handling behavior leads to high complexity and a violation of control link reversal expected in workflow rollback. We show how using only explicit handlers simplifies the model making it more usable, leads to less surprise behavior, makes calculating compensation order more efficient, and allows one to relax certain restrictions that WS-BPEL places on allowed constructs.

The over-loaded functionality of scopes leads to complexity in handling compensation. For example, a scope added just to scope a variable will by default affect the fault and compensation handling behavior of the process. A scope will always run its default compensation handler if: (1)a fault is thrown inside it and a matching fault handler is on one of its ancestor scopes, or (2)if it needs to terminate its nested scopes due to a fault thrown not within it but in one of its ancestor scopes. Several scopes can be compensated by a single handler by running their compensation handlers in some order. This order may be either an explicit order defined by the process designer or the 'default order' calculated according to the WS-BPEL standard. The latter is the prescribed order for a scope's default compensation handler. The default order is conceptually aimed at occurring in the reverse order of scope completion. However, calculating the reverse order in the presence of both scope nesting and links that cross scope boundaries is not straightforward. In WS-BPEL 2.0, the manner in which this order is calculated at runtime is made complex due to the interleaving of compensation, termination and fault handling as a fault is being propagated up the scope hierarchy to a matching fault handler. While the standard provides enough information for the resulting compensation order to be calculated at runtime, we doubt a process designer can take all the possibilities into account whenever adding a scope to a process. Examples of matters that complicate this order include that (1)a fault always propagates one level of scope nesting at a time and is re-thrown by the default fault handler which first terminates and compensates immediately nested scopes and (2)one must handle child scopes that were still running when the fault was caught before compensating completed ones.

A key goal of this work is to provide a more deterministic, simplified model for determining the default compensation order without sacrificing key functionality for workflow languages such as WS-BPEL that support a mix of graph and structured control constructs. We aim to reduce the large number of possible orderings created by the current behavior, which is highly sensitive to runtime state. Our approach is based on the concept of considering only the view of the world from the scope where the fault can be handled. This entails removing default fault and compensation handlers from WS-BPEL, as well as default and explicit termination handlers. Note that we propose removing 'default compensation handlers' which is not to be confused with 'default compensation order'. We keep the behavior that an explicit fault handler first

terminates all running activities inside its scope. Default compensation *order*, however, is kept only using the <compensate/> activity. In the absence of these default handlers, we propose a new compensation and fault handling scheme that is closer to what the designer expects and is still rooted in the same abstractions as WS-BPEL's.

The scenarios used in this paper are based on those brought forward by the WS-BPEL Technical Committee in OASIS (WS-BPEL TC) [21] as drivers for the current compensation order behavior in WS-BPEL. We compare the current behavior with the behavior that would result using our approach. In fact, the current WS-BPEL behavior is too restrictive because it does not allow a process to have 'peer-scope cycles', which are themselves included in some of the committee's own motivating scenarios. A 'peer-scope-cycle' is a cycle created by links between two scopes nested in the same parent scope. We will show how this restriction can be eased in our approach so that such scenarios can be constructed by workflow designers.

The rest of this paper is structured as follows: related work is followed by a presentation and illustration of WS-BPEL's current compensation behavior and the problems it presents. Then, an alternative approach is presented leading to the conclusion and future work.

2 Related Work on Recovery in Workflow

WS-BPEL combines both structured nested activities with explicit control dependencies in the form of links, combining the block structured (calculus based) and graph-based approaches of earlier workflow languages [13]. This has presented new challenges in adapting known compensation models to suit this change.

The concept of compensation scopes (known at the time as 'compensation spheres') was introduced for graph–based flow systems such as the model defined in [14,15]. A compensation sphere is a view on the process, grouping together a set of constructs with corresponding properties. Default compensation order was determined by reversing the control edges between the process's activities. However unlike WS-BPEL, such graph-based flow models did not include scope-level fault handlers and the scopes were not first class constructs with a place in the workflow's navigation. That is, spheres could not be the sources or targets of control links. Du et. al [7] presented a mechanism for automatically determining compensation scope boundaries at runtime based on dependency analysis, state, and user hints to provide more flexibility beyond requiring statically defining a scope at design time. In contrast, WS-BPEL's scopes are always defined at design time because they have a place in the navigation.

The need to rollback long running transactions, for which ACID is impractical, led to the creation of Sagas [9]. A Saga is made of several tasks, each being an ACID transaction and having an optional compensator which defines undo actions for that task. In the case the Saga needs to be aborted, then already running tasks are aborted and those that had completed have their compensators run in reverse order of completion. Nested Sagas [10] extended the model, so a task in a Saga could be either an ACID transaction or another Saga. Considering the scopes in a WS-BPEL process as representing a nested Saga, one can see a parallel between Sagas and WS-BPEL's default compensation and termination behavior today. However, the tasks in a Saga do not have the complex

relationships between them beyond nesting that WS-BPEL scopes do and neither do they combine fault handling as a different and complimentary capability to rollback.

It was shown in [1] that while the model presented by Sagas is too limited for a workflow environment, most advanced transaction models could be modeled directly into a workflow using patterns of existing workflow constructs. This was used in the Exotica project to provide a pre-processor that took a high level specification of a transaction model and generated the corresponding workflow artifacts to implement it. YAWL [22] is one example of a flow language which does not support compensation directly but in which one could model compensation behavior using patterns of YAWL constructs. [2] defines such a mapping from WS-BPELs compensation model to a YAWL 'compensation pattern'. They cover only explicit compensation, calling default WS-BPEL compensation 'troublesome'.

We conclude from the above that the combination of nested scopes as first class constructs in a workflow with the merging of calculus and graph based processes required additional work beyond existing rollback models. The result was today's WS-BPEL default compensation and termination behavior that combines the concepts from nested Sagas with those from compensation spheres in [15].

There is a large body of work on the use of transaction and coordination in the context of workflows and long running activities. We focus on the part most relevant to WS-BPEL itself and that would provide the context as to what the new problems are that it introduced and how this leads to the conclusion just presented. We do refer the interested reader to [6] for a more in-depth look at these models.

New research on the WS-BPEL compensation model itself has focused on requesting new features and on the explicit compensation case as opposed to default order and hence would not be adversely affected by our approach. In [3], the author describes WS-BPEL's compensation mechanism and requests a richer capability enabling a compensation handler access to current process state as well as the ability to call a compensation handler from outside of a fault handler. The former has already been done in WS-BPEL 2.0, while the latter is still not possible. In [11], the authors provide a look at workflow and transaction models, present a classification of the types of cancellation and recovery that are common in workflows and a statement of which of these can be supported by WS-BPEL directly. Some of the critiques of WS-BPEL's compensation capabilities have been addressed since then in WS-BPEL 2.0. They present an argument for having more flexibility in defining compensation and present a plan for creating a consistency language and tools to help with richer rollback capabilities. It would be interesting to see whether these would layer on top of the existing WS-BPEL compensation capabilities or perhaps make use of the extraction of compensation to an external coordinator done in [19].

Transaction models that go beyond compensation are overlaid on WS-BPEL processes to govern sets of interactions with external partners in [8] and [20]. Where [8] directly added transaction management capabilities using WS-BPEL language extensions, Tai et. al [20] addressed the problem by using policies, on existing WS-BPEL constructs, that are related to WS-BusinessActivity [17] and WS-AtomicTransaction [16]. A more detailed look into the relationship between WS-BPEL compensation and WS-BusinessActivity is presented in [19], where the main focus is to externalize the

coordination taking place between a scope and its child scopes by using an extended version of the WS-BusinessActivity protocol.

We had examined the intricacies of WS-BPEL's fault handling due to its combination of structured and graph–based flow models in [4]. We now do the same for compensation handling while also proposing an approach (based on section 3.6 of [12]) to simplify the default compensation order.

3 Current WS-BPEL 2.0 Compensation Processing

In this section, we describe how compensation handling occurs in the WS-BPEL standard. Compensation processing in WS-BPEL is based on the notion of scopes. A WS-BPEL scope is an activity that itself contains a group of activities, which themselves may also be scopes. Thus, scopes may be nested. The scope assigns certain characteristics to the activities nested within it. The entire process, by default, is a scope.

WS-BPEL provides mechanisms for trying to recover from faulty situations. Central to these mechanisms are 'fault handlers' that are attached to a scope and can catch and deal with faults in that scope. There are several ways in which a fault may be thrown in a WS-BPEL process, e.g.: A Web service called from a process instance may respond with a fault message or a process itself might detect erroneous situations that result in internal faults. When a fault occurs, the regular processing in the scope in which the fault occurred is interrupted and the fault is passed to a fault handler of that scope. The fault handler aims to correct the situation such that regular processing can continue outside the scope or alternate ways to complete the process can be taken. All of this might require undoing actions that have already been completed within the scope. Actions required to undo already completed activities are defined via 'compensation handlers'. Thus, a fault handler of a scope may start the compensation handlers of its nested completed scopes to undo their actions. It does so via a <compensate> activity. The compensate activity may optionally name a particular scope. If it names a scope, only the compensation handler associated with the specified scope is called; otherwise, it performs compensation on all enclosed completed scopes according to the default compensation order. A 'termination handler' aborts a running scope. A handler (fault, compensation, or termination)of a scope may contain any type of activity, including the empty activity. If any of the compensation or termination handlers is not specified for a scope, a corresponding *default handler* is provided. For fault handlers, a default fault handler is always provided on every scope to catch all faults except those for which an explicit fault handler is defined.

Figure 1 helps illustrate the details of the WS-BPEL fault handling and compensation mechanism. All the figures in this paper use the visual cues shown in figure 1's key. As shown, each scope is associated with three handlers: a fault handler, a compensation handler, and a termination handler. When navigation through the process enters a scope, the fault handler and the termination handler are installed. When the scope has completed processing, the termination handler is de-installed and the compensation handler is installed.

Figure 1 shows the following situation: the outer scope G is running with its fault and termination handlers active. Enclosed are scope F, which is still running with the

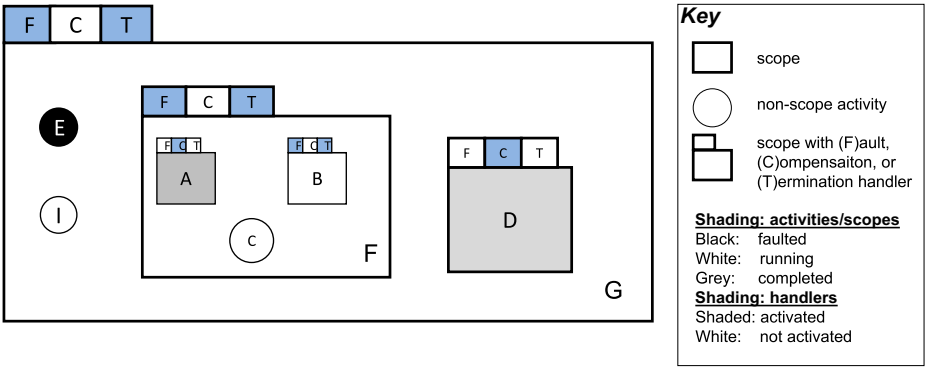

Fig. 1. Scopes With Associated Handlers

fault handler and the termination handler active, scope D which has completed, so its compensation handler is active, and the non-scope activity E, which has just faulted. Scope F has enclosed two scopes: scope A which has completed, scope B which is running. Scope F also encloses non-scope activity C which is running.

Fault handling is always initiated through an error raised in the flow. Each fault is associated with some fault code. When a fault occurs, all running activities within the scope in which the fault occurs are terminated. If the activity is not a scope, it is terminated according to the WS-BPEL specifications (wait for it to complete or simply abort it). If the activity is a scope, the associated fault handler is de-activated and the termination handler is carried out. This causes the following, shown in figure 1 : when activity E in figure 1 faults, first the running activity I is aborted. Next, the fault handler of scope F is de-installed and its termination handler is invoked.

The termination handler performs the following on its nested activities, in this order:

1. Enclosed non-scope activities: stop them if they are running. Otherwise, do nothing. This is done in arbitrary order.
2. Enclosed running scopes: deactivate their fault handlers and invoke their termination handlers. This is done in arbitrary order.
3. Enclosed completed scopes: invoke their compensation handlers. This is done according to the default compensation order. This is equivalent to performing a single <compensate> activity in the termination handler.

Thus, when the termination handler associated with scope F is invoked: First it terminates the non-scope activity C. Next, it processes the running scope B by de-activating the fault handler and invoking the termination handler. Last, it invokes the compensation handler of scope A.

Following termination, the matching fault handler is invoked. The default fault handler is used if an explicit matching one is not found on the current scope. In the case of figure 1, this means invoking the fault handler of G which in this case is a default fault handler. In WS-BPEL, a *default fault handler* runs a compensate activity and then rethrows the fault up to its parent (i.e. enclosing) scope. Thus, the fault handler of scope G

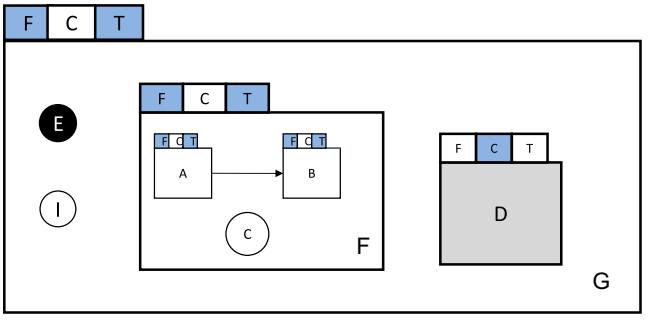

Fig. 2. Arbitrary Termination Sequence

therefore first invokes the compensation handlers of its nested completed scopes: here, scope D.

A scope's compensation handler usually contains the set of activities that undo the effects of the completed scope. For example, a scope that performs a payment commonly has a compensation handler that causes a refund. The *default compensation handler* runs a compensate activity, which causes the compensation handlers of all completed enclosed scopes to be invoked in default compensation order.

It is important to highlight that the described processing of fault, compensation, and termination handling causes first the running scopes (and thus all their enclosed scopes) to be processed before processing the already completed scopes. The processing of *running* scopes is in arbitrary order.

If, for example, as in Figure 2, two peer scopes are still running, then WS-BPEL does not prescribe any sequence in which the two scopes are to be terminated, even if they are connected via a control link. Thus, the WS-BPEL engine may first invoke the termination handler of scope A before invoking the termination handler of scope B. This may introduce some non-determinism; however, we can safely ignore this because termination in itself is somewhat non-deterministic (dependent on the state of the activity it may or may not be terminated).

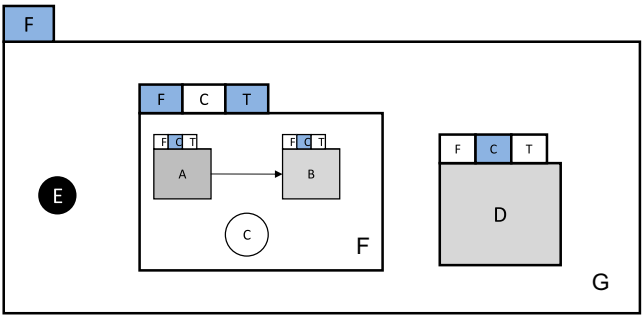

Fig. 3. Honoring Control Links during Compensation

Figure 3 shows the situation where the two peer scopes A and B have already completed. In this case, compensation honors the control link dependency between scopes A and B by first invoking the compensation handler of scope B before that of scope A.

The WS-BPEL specification mandates that the control links are honored when compensating peer scopes that have completed and that have a control link between them. This is the backbone of the concept of default compensation order. However, it violates this for non-peer scopes.

4 Problems with the Current Compensation Mechanism

The fault and compensation mechanisms suffer two main problems:

1. In the presence of control links that cross scope boundaries, the compensation order can violate the control link dependencies. An illustration of this problem is provided in section 4.1, showing that even for a simple case the default handlers of a scope may have devastating effects on the business logic and that WS-BPEL does not providing guarantees on compensation order between non-peer scopes.
2. A *zigzag* behavior of compensation is introduced by the default handling mechanism when scopes on different levels are being compensated, because compensation, termination and fault handling act on only one level of scope nesting at a time. An illustration of this behavior is provided in section 4.2.

The complexity of compensation and in particular the zigzag behavior is a problem because it is hard for process modelers, testers, and business analysts to comprehend the behavior a process will have due to compensation: they would have to always keep in mind all the current states in all the different scopes and their dependencies. The problem manifests itself in the modeling and particularly testing of workflows: one must really dive deeply into the process to understand the subtleties. One aim of our approach is to simplify the design and analysis of processes for our target audience of process modelers, testers, and business analysts. We do so by only performing compensation when explicitly requested by the designer and providing one possible default compensation order for each scope in the process. As a result, one can easily view and reason about that order to determine how best to design, refactor, or analyze one's processes.

4.1 Illustrating the Anomalies

We illustrate the example using the scenario in figure 4. This example is similar to the one in [5] and is an extension to figure 1. A control link has been added from scope B to scope D. It should be noted that this is only possible if the scope F is a non-isolated scope, otherwise the link would be blocked until scope F has completed. Furthermore all fault and termination handlers are default handlers. This case exhibits a compensation order which violates reversal of the control prescribed by the control link.

First, we provide a high-level view of part of this sample to show a concrete case of the problem and then perform a step by step walk through for the entire sample. Assume that the sample is for a service to special order an item and uses compensation for an error that requires a recall. Assume scope B processes the payment and D ships

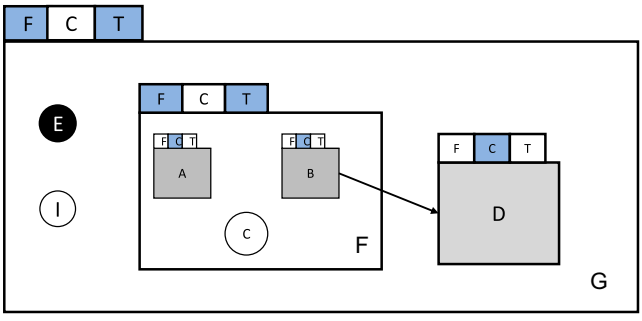

Fig. 4. Wrong Compensation Sequence

the product and waits for the customer to acknowledge receipt. Both B and D have explicit compensation handlers defined. The compensation handler of B refunds the customer. The compensation handler of D requests that the customer send the item back and wait until the merchant receives and checks the returned item. Assume that scope F is added to supply a shared, local variable 'creditCard' between customer-specific payment activities A, B, and C that shadows another process-wide 'creditCard' variable that stores corporate billing information. We stated that compensation is expected to occur so as to reverse control dependencies and thus the process would be expected to compensate D then B: first return the item and have it checked by the merchant then refund the customer. The next paragraph will show that for some (and not all) executions of even this simple process, WS-BPEL will compensate B before D which would cause the customer to be refunded before the item is returned by the customer and checked by the merchant. Some may argue that perhaps a solution is to redesign the process. However, not only would the reasons and options be confusing for the designer but even this simple example with a handful of scopes and two levels of nesting illustrates key problems: default handlers of a scope may have devastating effects on the business logic and there are no guarantees by the engine on compensation order between non-peer scopes such as B and D.

Now we consider the step by step runtime behavior of the sample in figure 4 when activity E faults. At that time, termination processing is initiated as follows. First the non-scope activity I is terminated. Next the fault handler of scope F is deactivated and the termination handler is invoked. The termination handler first terminates the running non-scope activity C, then it invokes the compensation handlers of scope A and B in default compensation order. Since A and B are not joined by a control dependency, then the default order is simply any order. When the two compensation handlers have finished, control goes to the fault handler of scope G. The fault handler, since it is the default fault handler, invokes a compensate activity which causes the compensation handler of scope D to be invoked.

This sequence of events shows that WS-BPEL carries out compensation (A then B or B then A, followed by D) violating the basic concept of reverse control order as determined by the control links between completed scopes.

4.2 Complexity of WS-BPEL2.0 Compensation Behavior

We have claimed that the current default handler behavior causes high complexity in the default compensation order making it difficult for a designer to anticipate the resulting behaviors when making process design decisions. In this section, we back this claim.

The following example shows the complexity involved due to the existence of default handlers. Notice in particular the 'zigzag' behavior as scopes from different levels in the hierarchy get compensated.

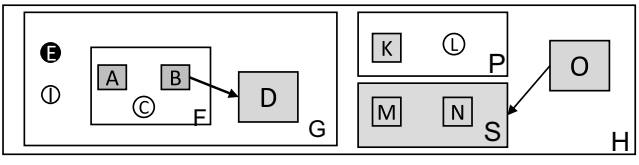

Fig. 5. Example of Compensation Complexity. Assume all scope have default FCT handlers.

In the example in Figure 5, we nest scope G from figure 4 within scope H along with three other scopes P, S, and O, which themselves contain scopes K, M, and N. The same behavior as described in the previous section above happens first, ending with the compensation of D. This is shown as steps 1 through 5 in figure 6 where the fault, compensation and termination of scopes is shown.

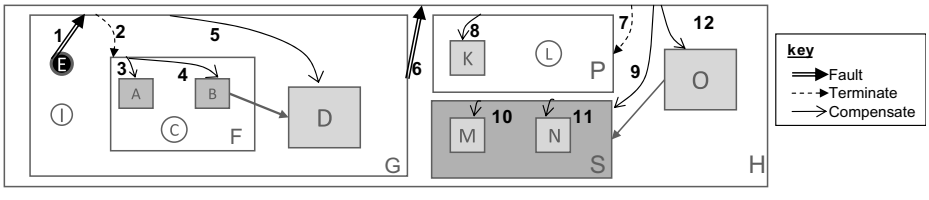

Fig. 6. Runtime FCT handling of scopes

To illustrate the behavior of what happens next, the rest of the description will refer in parenthesis to the step numbers on the arrows in figure 6: (6)the default fault handler of scope G rethrows the fault up to scope H. Recall that upon receiving a fault, a scope without a matching fault handler first terminates its running children and then runs its default fault handler. The default fault handler compensates completed immediate child scopes and finally rethrows the fault to its parent scope.

This is illustrated for scope H: it first terminates its running child P (7). P's default termination handler terminates L and runs the compensation handler of K (8).

Termination of P completes at this point. The default fault handler of scope H then compensates the immediate children of scope H in default order. This means reversing control order between peer scopes, i.e. S then O: First, the default compensation handler of scope S is invoked (9). This invokes the compensation handlers of scopes M (10) and N (11). Finally the compensation handler of scope O is called (12). When this handler

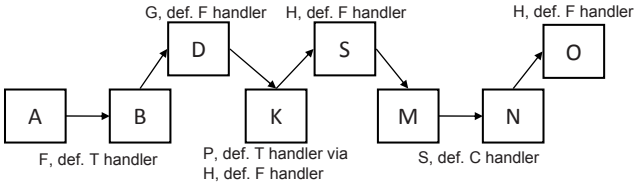

Fig. 7. WS-BPEL 2.0 Compensation order and triggers for scope H

completes, the default fault handler of scope H rethrows the fault to its parent scope (not shown).

As can be seen, compensation happens by first going down each of the scopes, compensating the innermost scopes first. Figure 7 illustrates the resulting order in which the compensation handlers of the shown scopes are called along with the respective handler that triggered the compensation. The zigzag behavior of the control leading to compensation is illustrated in Figures 7 and 6: from G to F, down to A and B, up to G, down to D, up to H, down to P then to K, back up to H down to S, down to M and N, and finally back up to O. In our approach, we avoid the zigzag behavior by immediately going to the scope with the matching explicit fault handler and performing the fault handling and optionally compensation as well on all nested scopes directly from that one matching scope, avoiding all these levels of indirection. Even though we remove default handlers, process designers will still be able to create more advanced compensation and fault handling patterns through the use of explicit handlers.

5 An Approach That Modifies Handler Behavior

We propose reducing the complexity of compensation processing and removing the anomalies presented in the previous sections by (1)Eliminating default fault handlers while specifying that if a fault reaches the top most scope (i.e. the process) without being caught then the process is terminated without any compensation, (2)Eliminating default compensation handlers as well as both default and explicit termination handlers, and (3)Modifying the reachability of compensation handlers such that a compensation handler of a scope may be called from any ancestor scope instead of only from the immediate parent scope.

The corresponding runtime behavior for our approach is as follows:

1. If a fault occurs in a scope that has no explicit fault handler for the particular fault, the fault is automatically propagated to outer scopes until an explicit matching fault handler is found. If no fault handler is found, the process is terminated. Otherwise, all running activities within the fault handler's scope are terminated (with no special termination handling).
2. Then, the fault handler is run.
3. If the fault handler calls default compensation via the <compensate> activity, then nested scopes are compensated according to the default compensation order. This order is dictated by the scope's 'Compensation Order Graph' (COG), constructed according to the detailed algorithm in section 5.1.

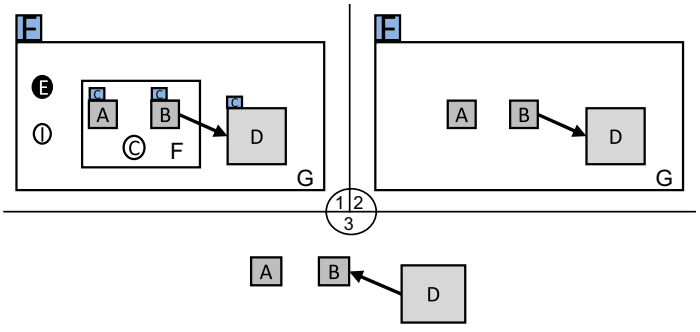

Fig. 8. Compensation of G: (1)scope G, (2)the view of relevant scopes, (3)the default compensation order in a COG

Note that the approach does not change the WS-BPEL behavior in which a fault not explicitly caught at any level in the process results in terminating the process without any requirement of notification. While this is reminiscent of silent termination, it is expected that such cases are reported to the process owner or administrator by implementation specific means and are thus left out of the standard (and therefore the modifications presented in this paper). WS-BPEL by design does not enforce management or administration behavior, leaving those up to WS-BPEL engine vendors.

An important part of the COG's construction is its treatment of scopes according to the presence of explicit compensation handlers: if a scope has no explicit compensation handler, the scope does not exist from a compensation perspective, it is *transparent*; if a scope has an explicit compensation handler, then the inner structure is not known to the compensation mechanism, that means the scope is *opaque*. Note that this meaning of opaque is unrelated to the use of opaque in Abstract WS-BPEL Processes.

In addition to respecting linking behavior, COGs are calculated at design time and always dictate the order for default compensation, thereby allowing a designer to study the (small set of) possible execution sequences of the process. In contrast, the behavior in WS-BPEL today results in sequences that will differ at runtime based not only on scope nesting and linking but also on scope completion times and termination handler behavior as shown in section 4. In WS-BPEL today, one can still create graphs at design time to encode compensation order (section 5.5.3 in [12]); however, the default order is not always run according to these graphs because (1)they represent one level of scope nesting at a time and (2)WS-BPEL dictates that active immediate child scopes are terminated (and their children possibly compensated) before compensation of completed immediate child scopes.

In order to illustrate our approach, consider that the fault handler in figure 4 has been defined explicitly and just carries out compensation through running a compensate activity. Since our approach centers on dropping default handlers, explicit compensation handlers on relevant scopes as shown in figure 8(1). Then the compensation processing would see the situation as shown in figure 8(1). Since F is transparent, it is obvious that the control links are honored for compensation.

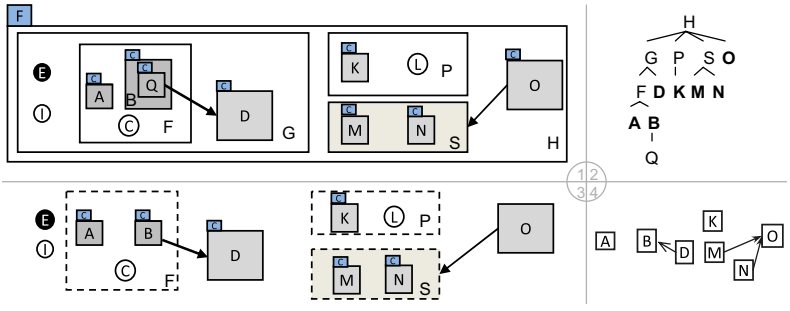

Fig. 9. COG of H: (1)scope H, (2)the scope-and-loop hierarchy subtree rooted at H, (3)determining COG edges, (4)the COG of H

Figure 9(1) shows a sample scope H and the steps, described in the next section, for determining its compensation order and encoding it in a Compensation Order Graph (figure 9(4)). Scope H is similar to the scope in figure 7 but has explicit compensation handlers, no default handlers, and changes the source of the link from scope B to D to be from scope Q in B. An interesting point is the necessity in the COG to draw control links between scopes M and N and scope O as will be described in section 5.1.

5.1 Building the Compensation Order Graph

The Compensation Order Graph (COG) of a scope is used to calculate the default compensation order for that scope; that is, the order in which explicit compensation handlers of nested scopes must be run. In our approach the only way that compensation in default order gets triggered is by a compensate activity in a fault handler. Thus, the COG need only be calculated for scopes that have at least one explicit fault handler containing a compensate activity. In contrast, any scope in a standard WS-BPEL process may have to perform default compensation.

Conceptually, the COG of a scope is the graph formed when looking down from that particular scope at scopes nested within it at any level, such that opaque scopes are collapsed to a single node and transparent scopes are ignored without obscuring nested opaque scopes from view. The opaque scopes thus found form the COG's nodes. The edges between the nodes reverse the control dependencies present in the process between the corresponding scopes. In reality, it is slightly more complicated because one must also handle scopes nested in loops and links that cross scope boundaries as will be shown in the rest of this section. We first present background on the treatment of control dependencies and then detail the algorithm for constructing the COG of a scope, starting with determining COG nodes and proceeding to COG edges.

The edges of a COG are based on the control dependencies in the process between the corresponding scopes, including both: dependencies specified by explicit control links and the implicit dependencies imposed by structured activities like 'flow', 'sequence', and 'while'. WS-BPEL does not allow control links to cross the boundary of the looping structured activity 'while', nor does it allow explicit control links to create cycles.

Having presented the intuition behind COGs, we now present the algorithm for creating them. Consider a loop in the process to be a 'compensation relevant' loop of scope s if it (1)is nested in s, (2)not nested in any child scope of s, and (3)contains, at any level of nesting, at least one immediate child scope of s that has an explicit compensation handler and is not in a fault handler of s.

Consider a 'scope-and-loop hierarchy tree' whose nodes are the scopes and loops of the process. Each loop or scope has an edge to its immediately enclosing loop or scope. For example, if a scope s_2 is in loop l_1, l_1 is in another loop l_2, and l_2 is in a second scope s_1, then the tree's edges would be $(s_2, l_1), (l_1, l_2), (l_2, s_1)$. Figure 9(2) shows the scope and loop hierarchy subtree rooted at scope H. The COG nodes (in bold in the figure) are found by navigating this tree as follows:

Consider $N_{COG}(s)$ to be the set of nodes of the COG of s. Then, $N_{COG}(s) = S_{COG}(s) \cup L_{COG}(s)$, where $S_{COG}(s)$ is a subset of the scopes nested in s and $L_{COG}(s)$ is a subset of the compensation relevant loops of s. The nodes are determined by walking the scope-and-loop hierarchy sub-tree rooted at s in a depth first manner. For each visited node, if it (1)is a scope, (2)has an explicit compensation handler, and (3)is not in a fault handler of s, then it is added to $S_{COG}(s)$. Alternatively, if it is a compensation relevant loop of s, then it is added to $L_{COG}(s)$. Otherwise, it is ignored. During the depth first traversal, the children of a node added to $S_{COG}(s)$ or $L_{COG}(s)$ are *not* visited. Thus, $N_{COG}(H) = S_{COG}(H) = \{A, B, D, K, M, N, O\}$. Notice that Q is not visited.

For each loop l in $L_{COG}(s)$, create a corresponding 'loopCOG'. A loopCOG's nodes and edges are constructed the same way as for a normal COG except starting from the scope-and-loop hierarchy sub-tree rooted at l.

Determining the edges for the COG of a scope s requires determining the control dependencies between the COG's nodes. To do so, perform the following on the activities in the process nested in s. (Any changes to process structure described in this section are for the purposes of COG construction only and must not be reflected in the workflow designer's actual process.)

1. Collapse each loop l that belongs to $N_{COG}(s)$ into one node by hiding activities nested within it. Recall that no links may cross the boundary of a WS-BPEL loop so no dangling links will result from collapsing l.
2. Collapse each 'opaque scope' s_o that belongs to $N_{COG}(s)$ into one node by hiding activities nested within it. Handle any dangling links that result by moving the target/source of each link crossing the boundary of s_o to have s_o as its target/source. More precisely, if the source activity of a link is nested in s_o but its target is not, then treat s_o itself as the link's source; if the target of a link is nested in s_o and its source is not, then treat s_o itself as the link's target.

   The result of this step is shown in figure 9(3): scope B is made the source of the link from scope Q (nested in B) to scope D. In the figure, the transparent scopes P, F, G, and S are shown with dashed lines to illustrate that they are ignored in the COG since they are not among its nodes. Notice that activities such as M and N that are nested in transparent scopes, however, may appear in the COG.
3. Create the edges: if after these modifications there exists a control path in the process between a node n_1 in the COG (or any activity in n_1) to a node n_2 in the COG

(or any activity in n_2), such that the path does not contain any other COG node n_3 (or activity in n_3), then an edge (n_2, n_1) is added to the set of edges of the COG.

The only relevance of transparent scopes to the COG is that they may impose implicit control dependencies that form segments of paths between COG nodes, such as the control paths from scope node O to M and to N via the transparent scope S. This path is due to the nesting of M and N in S, in which case WS-BPEL semantics dictate that there is an implicit control dependency such that M and N may only start if S has started. One way to model both implicit and explicit dependencies in order to determine paths is to flatten the process into a graph-based model using the algorithms in ([12], Chapter 3.4), whose details are out of scope for this paper. Figure 9(4) shows the resulting COG of scope H, including the edges, while figure 8(3) shows the COG of G.

The third step above reverses the dependencies in the process between COG nodes. For example, the COG of G contains the edge (D, B) between the COG nodes representing scopes B and D, whereas the process itself had a link from Q in B to D.

One must also create, for each loop node l in the COG, the edges of the corresponding loopCOG. This is done in the same manner as for the edges of a COG, but on the activities in the process nested in l and resulting in edges between the nodes of the loopCOG.

No cycles are allowed in any of the generated compensation graphs. This is a relaxation of the current WS-BPEL restriction, mentioned in section 1, disallowing peer-scope-cycles regardless of whether and where explicit compensation handlers are used. The main reasons why this is a relaxation are that COGs are not created for every scope and not every scope appears as a node in the COG of another scope. Thus some peer scopes cycles are allowed when using our approach, such as scopes that do not have compensation handlers or those that do but are not in the COG of another scope.

5.2 Running Compensation According to the Default Order

Having described how the COGs are created to encode the default compensation order, we now explain how compensation in default order can be executed by navigating the COGs at runtime.

When compensation of a scope in the default order is triggered, the process engine navigates the scope's COG. For the purposes of navigation, each COG can in fact be treated as a process itself where (1)the transition and join conditions, which are WS-BPEL constructs that enable conditional branching and joining, are set to true, (2)the implementation of each node created from collapsing a scope results in a call to the compensation handler of the corresponding scope, and (3)the implementation of each node created from collapsing a loop corresponds in navigating (possibly more than once) the loopCOG of that node.

The number of times to run the COG of a loop depends on the number of times the loop ran in the instance of the scope being compensated. This number of iterations, along with information for keeping track of scope instances and whether or not they can be compensated, requires book-keeping. The details of this book-keeping are left out of this paper because it is also required in standard WS-BPEL (without our modifications). The interested reader is referred to Section 3.6.3 in [12] for details of a structure for this purpose named the 'ScopeLoopQ'.

6 Advantages and Drawbacks

In this section, we summarize how the new approach meets our goals of simplifying designing, refactoring and analyzing processes with regards to default compensation order as well as respecting control order. We lay out the gains and highlight a few drawbacks. We have shown the following advantages of this approach:

- Simpler for designers, testers, and business analysts to design and analyze their processes. The compensation order is easy to understand and visualize, consequently simplifying the design process. It is also only triggered explicitly, making it more predictable. This simplification could lead to richer process semantics due to providing the designer with more control over what will take place at runtime. Whereas the default compensation order in BPEL today is a function of which activity raised a fault, which scope it was caught in, and the levels of scope nesting in between, we have shown a default compensation order that is simply a function of the process structure and can be calculated at design time and presented to the designer.
- Enables separation of concerns between different capabilities of a scope. For example, a designer who adds a scope in order to simply scope a variable will not see a change in resulting compensation order at runtime. If, however, the designer does want such a change, she may simply add explicit fault and compensation handlers.
- Provides more efficient execution due to the ability to calculate the COGs before runtime and avoid the zigzag behavior. In WS-BPEL today, the order can only be determined at runtime and involves several levels of scope nesting. Additionally, we know apriori which subset of the process's scopes may be compensated and calculate the default compensation order only for that subset.
- Respects the order specified by the explicit process control flow and performs repairs at the level where the fault is caught. WS-BPEL today exhibits some cases where default compensation order is in contradiction with order specified by reversing control edges.
- Loosens the WS-BPEL restriction on cycles between peer scopes.
- Easily injectable into WS-BPEL engines because it is a variation on the existing compensation mechanism.

The drawbacks include the removal of custom termination handling and the deviation from the WS-BPEL standard so this behavior cannot be added as a WS-BPEL extension. However, termination handlers have not been a common feature in workflow systems, so it is unclear that this will be a major problem for process designers. On the other hand, it would be possible for our approach to allow termination handlers as long as they are explicit and disallowed from triggering compensation.

7 Conclusion and Future Work

In this paper, we investigated how the advent in process models of combining structured and graph based process modeling techniques, as well as a combination of fault and compensation handling, affects the roll-back and recovery capabilities of a workflow language. We framed the discussion in context of existing approaches to workflow

recovery and prior art in nested transactions. Focusing on WS-BPEL, we provided an in–depth look into WS-BPEL's compensation mechanisms with a focus on default compensation, showed where it causes complications and possibly unexpected results, and provided an alternate approach for calculating the default compensation order that is based on the view from the scope where a fault will be explicitly handled. We also advocated for the removal of default fault and compensation handlers as well as termination handlers. The proposed approach presents several advantages that we illustrated, including a deterministic order for default compensation of scopes and decoupling the compensation and fault handling behaviors of scopes from each other and from other scope functions such as variable scoping. This leads to less complexity and less surprise for the user. It also enables one to relax the peer-scope dependency restriction currently in the standard.

We continue to investigate several open areas such as allowing an opaque scope to be treated as transparent if it did not successfully complete at the time it became a candidate for compensation and the effect on event handlers. A longer term research goal is to explore the ramifications of this approach beyond workflow and WS-BPEL; for example, to determine whether it could augment existing transaction models such as nested Sagas or provide a new transaction model generalized beyond WS-BPEL.

References

1. Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Gunthor, R., Mohan, C.: Advanced transaction models in workflow contexts. In: Int'l Conference on Data Engineering (1996)
2. Brogi, A., Popescu, R.: From BPEL Processes to YAWL Workflows. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 107–122. Springer, Heidelberg (2006)
3. Coleman, J.: Examining BPEL's compensation construct. In: Workshop on Rigorous Engineering of Fault-Tolerant Systems, REFT (2005)
4. Curbera, F., Khalaf, R., Leymann, F., Weerawarana, S.: Exception handling in the BPEL4WS language. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 276–290. Springer, Heidelberg (2003)
5. König, D.: R26: Default Compensation Order Conflict (2006), `http://www.oasis-open.org/committees/download.php/21303/WS_BPEL_review_issues_list.html#IssueR26`, `http://www.oasis-open.org/committees/download.php/21199/Issue%20R26.ppt`
6. Dayal, U., Hsu, M., Ladin, R.: Business process coordination: State of the art, trends, and open issues. In: Very Large Databases Conference, VLDB 2001 (2001)
7. Du, W., Davis, J., Shan, M.-C.: Flexible specification of workflow compensation scopes. In: GROUP 1997: Proceedings of the international ACM SIGGROUP conference on Supporting group work. ACM, New York (1997)
8. Fletcher, T., Furniss, P., Green, A., Haugen, R.: BPEL and business transaction management (2003), `http://www.oasis-open.org/committees/download.php/3263/BPEL.and.Business.Transaction.Management.Choreology.Submission.html`
9. Garcia-Molina, H., Salem, K.: Sagas. Proc. ACM Sigmod (1987)
10. Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K.: Modeling long-running activities as nested sagas. IEEE Data Eng. Bull. 14(1) (1991)

11. Greenfield, P., Fekete, A., Jang, J., Kuo, D.: Compensation is not enough. In: International Conference on Enterprise Distributed Object Computing Conference (2003)
12. Khalaf, R.: Supporting Business Process Fragmentation While Maintaining Operational Semantics: A BPEL Perspective. PhD thesis, University of Stuttgart (2008), http://elib.uni-stuttgart.de/opus/volltexte/2008/3514/ ISBN 978-3-86624-344-6, dissertation.de
13. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise Modelling and Information Systems 4(1), 3–13 (2009)
14. Leymann, F.: Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems. In: Proc. BTW 1995. Springer, Berlin (1995)
15. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice-Hall, Upper Saddle River (2000)
16. OASIS. Web Services Atomic Transaction (WS-AtomicTransaction) version 1.1 (2007), http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec.pdf
17. OASIS. Web Services Business Activity (WS-BusinessActivity) version 1.1. (2007), http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec.pdf
18. OASIS. Web Services Business Process Execution Language (WS-BPEL) Version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
19. Pottinger, S., Mietzner, R., Leymann, F.: Coordinate BPEL Scopes and Processes by Extending the WS-Business Activity Framework. In: Meersman, R., Tari, Z. (eds.) CoopIS 2007. LNCS, vol. 4803, pp. 336–352. Springer, Heidelberg (2007)
20. Tai, S., Khalaf, R., Mikalsen, T.A.: Composition of coordinated web services. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 294–310. Springer, Heidelberg (2004)
21. Thatte, S., Roller, D.: Default compensation order (2003), http://www.oasis-open.org/committees/download.php/4449/Default%20Compensation%20Order.ppt
22. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. Inf. Syst. 30(4) (2005)