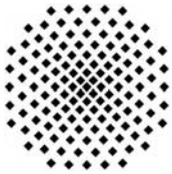

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Coordination Protocols for Split BPEL Loops and Scopes

Rania Khalaf, Frank Leymann

Report 2007/1
March 11, 2007

Institut für Architektur von Anwendungssystemen

Universitätsstr. 38
70569 Stuttgart
Germany

CR: C.2.4, D.2.6, D.2.11, D.2.12, H.4.1

Summary

The document presents an approach to enable loops and fault handling, compensating scopes to be split among a set of BPEL processes running on different BPEL engines. A mechanism to split a scope or loop into multiple fragments is presented, then a protocol is defined that can be used to coordinate fragments of a loop or a scope so that those fragments run as if they had been in a single process.

The requirements for running split scopes and loops are explained. For compensation, this paper focuses on explicit compensation and makes the assumption that compensation handing does not fail.

Two protocols are defined such that they may be plugged into the WS-Coordination framework. The messages between the participant fragments and the coordinator are defined. The information about the participating processes that the coordinator needs to have is specified. An algorithm is provided to locate a fault handler in the hierarchy of scopes that can handle a particular BPEL fault. Additionally, the behavior of both participants and the coordinator are specified.

Contents

1. Introduction

In an effort to ease business process partitioning and outsourcing, we have presented an approach [4] for arbitrarily splitting business processes based on business roles. In this approach, a set of activities in a business process, derived from BPEL[3], is assigned to a partner. Then, a mechanism was presented to show how one can generate one business process for each partner so that the cumulative behaviour is the same as that of the original project. In [4], we focused on processes expressed as flat graphs. In this paper, we add the capability to also split loops and scopes. For scopes, we focus on splitting the fault handling and compensation behaviour and not on event handlers or shared data.

On attempting to split loops and scopes using BPEL activities in the process (invokes and receives in certain composition patterns among the fragments), the amount of activities and the complexity of the resulting processes drastically increased. The complexity was mainly due to the fact that one needs to synchronize the start and end of all fragments that result from the split, as well as possible abortion, and rollback. For example, if one fragment of a loop completes then it has to wait for all others to complete and also notify them that it itself has completed. Therefore, we find that splitting loops and scopes is one clear point where it is worth bringing in additional capabilities to the BPEL language and augmenting the pure BPEL processes we have so far been able to maintain.

We propose solving the problem of splitting loops and scopes by adding a coordinator that coordinates the work of the fragments of loops and scopes so that they can behave as logical units. In order to lower the barrier of entry, we provide the coordination as protocols that plug into the WS-Coordination framework [2].

In this paper, we present an overview of WS-Coordination, describe the parts of WS-Coordination used to enable splitting processes, shows how to define fragments, and presents the coordination protocols required to execute split loops and fault-handling scopes.

We include split compensation handling, but only detail the case of explicit compensation at this time. Default compensation, although present in the protocol diagrams, will be out of scope for this paper and only briefly described.

It is assumed that the reader has a familiarity with BPEL.

2. Background: WS-Coordination

WS-Coordination is a pluggable framework for coordinating the agreement of the outcome of the execution of a collection of services that jointly perform a "distributed action". For that purpose, WS-Coordination specifies two middleware-related services: an Activation Service and a Registration Service. The Activation Service, which is optional, is used by the initiator to create a unique context identifying the distributed action to be started. Then, this context can be exchanged in the header of application messages between the services of the distributed action. Upon first receiving a message with a coordination context, such a service registers for participation in that distributed action with the Registration Service. Registration especially encompasses associating a "protocol handler" with each such service; a protocol handler is the entity in charge of exchanging messages to reach outcome agreement between the services. A distributed action is thus equivalent to an instance of the associated agreement protocol. The combination of registration and activation allows services to dynamically join such a running protocol instance.

Being a framework, WS-Coordination allows one to define one's own protocols and provide corresponding protocol handlers to perform the actual coordination logic. Defining a protocol consists of possibly extending the activation and/or registration messages, defining the port types of both the participants' and coordinator's protocol handlers, the order in which the protocol messages need to be exchanged, and the required underlying behavior as a result of receiving or sending a protocol message. For example, if a coordinator sends a *commit* message to a participant, the participant must respond with *committed* - but only after doing the work associated with the *commit* message (i.e.: committing the transaction).

When splitting processes each fragment of a split activity (loop or scope) corresponds to a service of a distributed action. In case of a split loop, for example, the associated agreement protocol is about agreeing whether or not another instance of the split loop body must be run. The fragments of a split activity are known at design time and their location is known at deployment time. Additionally, the fragments are peers: A fragment of a loop in one process can be reached before its associated other fragments in other processes, and it is not known ahead of time which one will start the corresponding protocol instance.

The case at hand in this paper differs slightly from traditional coordination as shown by protocols such as WS-BusinessActivity and WS-AtomicTransaction. Most importantly, the optional Activation Service is not used because a protocol instance is automatically created when the first

fragment of a split activity is reached and the context does not need to be dynamically flowed between the participants. Additionally, since the fragments were created to work together, each participant process has enough information for the coordinator to determine exactly which protocol instance it belongs to. The details of creating and using protocol instance identifiers will be explained in later sections.

3. Scope and Loop Fragments

A set of processes working in concert and needing to be coordinated together through our approach (usually because they were derived from a single larger process) are referred to as process fragments. The overall process formed by these fragments is referred to as a split process.

Loops and scopes can be split across a set of process fragments. The part of a split scope in one process fragment is referred to as a scope fragment. In other words, a split scope is made up of a set of scope fragments distributed between two or more process fragments. The same applies to a split loop and its constituent loop fragments. In order to be able to identify the fragments of the same split scope, we place the requirement that a split scope or a split loop must be uniquely named in the split process and that all fragments of the same split scope or loop will have this name.

In this work, the focus is on performing the behaviour of these artifacts as it relates to control and consistent lifecycle. Sharing data between fragments or control dependencies between entire activities (due to control links in the original process) are the domain of the work initiated in [4]. Such data and control dependencies are addressed using BPEL constructs in the fragments and not through the coordination. Therefore, we do not address them here.

In [4], we handled data dependencies that were expressed as 'data links' and are currently it to enable deriving and maintating the data dependencies expressed in BPEL's shared variable approach.

3.1 Summary of Restrictions

A few restrictions are assumed in this work. They are summarized in this section for quick reference. Sections providing more details are included in parenthesis.

1. One correlation set is shared among all process fragments. This is done using the process definition. In BPEL, correlation sets are used to determine instance identification information from application data. All messages arriving at a process must carry this information. (Sections 3.2, 3.4, and 6.2).

2. The join condition of a split loop or scope is restricted to a conjunction of the local join conditions of each fragment of a split loop or scope. The join condition is a Boolean valued expression defined on a BPEL activity. The activity is executed once control flows to it from its parent activity, all of its incoming links have fired, and its join condition evaluates to true. (Section 4).
3. This work does not address BPEL scopes' additional capabilities of having event handlers or scoping variables. Only fault handling and compensation handling capabilities of scopes are considered.
4. The loop condition in a split loop is the responsibility of exactly one fragment, instead of having the coordinator evaluate it from a set of variable values received from all or some of the fragments. In other words, we do not support splitting loop conditions. (Section 5).
5. The immediate child activity of a split loop or scope is restricted to be a <flow> activity; furthermore, this <flow> activity is disallowed from having incoming or outgoing links crossing its own boundaries. Its only purpose is to provide a container for the enclosed activities. Note that loops and scopes in BPEL must have exactly one immediate child activity.
6. Compensation is a recovery mechanism, and thus must not fail.

3.2 Identifying fragments

The process fragments of one split process must work together. More specifically, an instance of a fragment does not run in isolation. It must interact with instances of the other fragments of the split process. There may be several instances of each fragment running simultaneously. Therefore, one must be able to distinguish them and run the correct set (one instance per fragment) together. One way to address this problem natively in BPEL is to use the language's correlation mechanism.

In order to be able to correctly identify the instances of all the fragments of a split process that belong to the same 'run' of the split process, we provide a shared correlation set between all the fragments of the process. This is done in the process definitions. The value of this correlation set will be set be the message that creates an instance of a process fragment and will be used as the instance identifier.

3.3 Adding Coordination Protocols for Split Scopes

Next, we present several requirements resulting from our coordination approach to take into account the nature of the behaviour needed for splitting scopes and loops. The following steps need to be taken for that purpose:

- Extend BPEL to denote a scope or loop as a fragment.

    - Identifying the fragment enables one to signal to the BPEL engine in which the process will run that this activity must be coordinated and cannot be run as any other loop or scope.
- Provide interfaces for the participant and coordinator protocol services.
    - WS-Coordination requires a WSDL interface for each.
- Adapt the BPEL engine so it may be controlled as the protocols dictate.
    - The protocols require information from and need to be able to affect the process instances. For example, upon reaching a split loop or scope, the engine must wait for the remaining fragments of that loop or scope to start. Upon completing it, it must wait for the others to complete. Upon a fault being thrown in one fragment, the others must be stopped. Therefore, one needs a mechanism by which to control the running process based on messages from the coordination protocol.
- Provide the coordinator with information about the process model.
    - Each process fragment alone does not have enough information on the entire split process. It sees only part of the picture: that in which it is involved directly. For example, a fault handler may exist in one fragment of scope but not on the other fragment of the same scope. Such global information is placed, in our approach, in the coordinator to enable decisions to be made where information is needed about the process model that involves more than one fragment.
- Provide the logic at the participant and coordinator sides to implement the protocols.
    - The WSDL interfaces for the protocol must be implemented so that they send and receive the proper messages, as well as perform the appropriate expected behaviour both in the process and in the coordinator. For example, if one fragment of a split scope is ready to start then the participant side of the protocol implementation must send a *starting* message to the coordinator. Additionally, that scope must be blocked and not allowed it to run any of its activities until the coordinator sends it a *start* message. Furthermore, the coordinator implementation will not send the *start* message unless it has received *starting* from all fragments of this split scope.

3.3.1 The Rubber Band Effect

In BPEL, scopes and loops must be strictly nested. When split, this restriction is kept. It results in a property we define as the 'rubber-band' effect. Note that a process itself is treated as a scope in BPEL.

Corollary: If **S** is a split loop or scope, then all parent scopes of **S** (including the process itself) are split scopes.

Particularly, if *S* has fragments in partner A and partner B, then all parent scopes of *S* also have fragments in partner A and partner B. Consider a scope *S'* in a fault handler of scope *S",* where the immediate parent scope of *S'* is not in that fault handler. For the purposes of the rubber band effect, S' is treated as a child of *S"*. Therefore, *S'* cannot be split if *S"* is not split.

3.3.2 Deriving and Encoding Split Scope and Loop Relationships

The relationships between split scopes, their handlers, and their fragments, as well as between split loops and their fragments can be represented by the 'relationship tree' defined below.

The relationship tree, $RT=(N_{rt}, E_{rt})$, consists of a set of nodes N_{rt} and a set of edges E_{rt}. N_{rt} is divided into four pair wise disjoint sets, each containing a different kind of node. These sets are S_{rt}, S_{ns}, L_{rt}, and F_{rt}.

- S_{rt}, the set of split scope-nodes. Each scope node $s_{rt} \in S_{rt}$ is a tuple consisting of the name of the scope, a set M_s of names of faults that the scope has handlers for, a Boolean stating whether or not this scope is in the fault handler of its immediate parent scope, and a Boolean stating whether or not this scope is in the compensation handler of its immediate parent scope.

$$s_{rt} = (name, M_s, in\_fault\_handler, in\_comp\_handler) \quad (1)$$

- S_{ns}, the set of non-split compensation-relevant scopes. A non-split child scope node is *only* included in the tree if it or any of its nested scopes have an explicit compensation handler. If not, then it does not need to be coordinated and may safely be ignored.

  Each such node, $s_{ns} \in S_{ns}$ is a tuple consisting of the participant name and a Boolean denoting whether it has an explicit compensation handler. Such scopes are not participating in a protocol instance themselves.

$$s_{ns} = \{p\_name, name, has\_compensation\} \quad (2)$$

- L_{rt}, the set of split loop-nodes. Each loop node, $l_{rt} \in L_{rt}$, is a tuple consisting of the name of the loop and the name of the participant responsible or the loop condition.

$$l_{rt} = (name, condition\_p\_name) \quad (3)$$

- F_{rt}, the set of fragment-nodes. A fragment node represents one fragment of a split loop or scope. All fragment nodes are leaf nodes in the relationship tree. Fragment nodes are of two kinds, in the subsets of F_{rt} such that:

$$F_{rt} = F_{rt_s} \cup F_{rt_l} \tag{4}$$

- F_{rt_l} , the set of loop fragments. Each loop fragment node, $f_{rt_l} \in F_{rt_l}$, is a tuple consisting of the name of the participant the fragment is in (p_name), a Boolean stating whether that fragment is responsible for evaluating the condition of the loop, and the name of the loop to identify the loop that this fragment is part of.

$$f_{rt_l} = (p\_name, is\_responsible\_for\_condition, loop\_name) \tag{5}$$

- F_{rt_s} , the set of scope fragments. Each scope fragment-node, $f_{rt_s} \in F_{rt_s}$ is a tuple consisting of the name of the participant the fragment is in, a set O of fault names that this fragment of the scope (regardless of other fragments of the same scope) has handlers for, and the name of the scope to identify the scope that this fragment is a part of. The fault names must be a subset of the set M_s of the scope-node the fragment is part of. It also contains a Boolean stating whether or not at least one fragment of this scope has an explicitly defined compensation handler.

$$f_{rt_s} = (p\_name, O, scope\_name, has\_compensation) \text{ where } O \subseteq M_s \tag{6}$$

The tree contains three kinds of directed edges: $E_{rt} = A \cup B \cup C$. A, B, and C are pair wise disjoint.

- *A*, the set of fragment-activity edges. Each fragment-activity edge, *a*, connects either the fragments of a loop to its corresponding loop-node, or the fragments of a scope to its corresponding scope-node. Therefore, each fragment-activity edge is a tuple.

  Clearly, the fragments of a scope are always 'scope fragments' and the fragments of a loop are always 'loop fragments'. Therefore, scope fragments cannot be linked to a loop-node via a fragment-activity edge and vice versa. We define *A* as:

$$A \subseteq (F_{rt_s} \times S_{rt}) \cup (F_{rt_l} \times L_{rt}) \tag{7}$$

- *B*, the set of loop-scope edges. Each loop-scope edge, *b*, connects loops to their parent scopes. Each loop node is connected to exactly one scope-node with a loop-scope edge. Therefore,

$$B \subseteq L_{rt} \times S_{rt} \tag{8}$$

- *C*, the set of child-parent edges. An edge *c* in *C* directly represents the nesting (child-parent) relations between scopes. Scopes in BPEL are strictly nested, so each scope node has exactly one child-parent

edge to the scope node representing its immediately enclosing scope in the process. Therefore, a scope-node is connected to other scope-nodes by child-parent edges. A scope in a fault handler of another scope is considered the latter's child scope.

$$C \subseteq (S_{ns} \cup S_{rt}) \times S_{rt} \tag{9}$$

The protocols are designed such that the nesting of loops within other loops is not relevant. Therefore such nesting is not represented in the relationship tree.

The relationship tree can be derived automatically from the set of process definitions forming the split process, as long as scopes that are split are identifiable as such, all fragments of the same split scope or loop have the same name, and the nesting relationships are valid across the process fragments. In other words, consider that a scope fragment *s'* is a child scope of scope fragment *s* in one process fragment. Then, all other fragments *s'* must be children of all other fragments *s* across the processes.

It is of interest to note that if one were to support default compensation, the coordinator would be also be provided with a set of graphs encoding default compensation order.

3.4 Deployment

This section addresses the two parts needed to set up the connections between the processes and the coordinator: deployment and registration. As described at the end of section 2, no Activation Service is used for these protocols.

Deployment: The deployment descriptor provided in [4] for every split process is extended if it includes split scopes and/or loops, to also provide the endpoints of a default *starter service* for each participant. This service, detailed in section 6.2, is used to enable the coordinator to start process instances. The deployment descriptor also provides the address of the registration service of the coordinator so that the first process fragment to start an instance can go register for the split scope protocol. This address includes the name of the overall split process (as a WS-Addressing [1] reference property), needed in order for the coordinator to find the proper information related to this split process, such as the correct relationship tree, once a process registers.

Before being able to run the protocol for split scopes the coordinator must be aware of the *relationship tree* for the process model containing the split scope. We recommend passing this information as part of deployment of the process model.

Registration: WS-Coordination's Registration Service is in charge of establishing a connection between the environments hosting the various fragments. This connection is established based on "protocol handlers" being able to receive and process the corresponding protocol messages. When registering for a protocol, participants must provide a WS-Coordination protocol identifier identifying the protocol a participant will respect, and an address for the protocol handler of that participant. We define one identifier for the scope protocol and another for the loop protocol.

Three more pieces of information are needed to set up the connections: The coordinator needs to know the scope/loop name so it can determine which split activity this registration is for, the name of the participant for which this registration is for so it can determine which fragment is registering and match that to the information in its relationship tree, and an identifier to identify the instance of the coordinated work (the value of the common correlation set) so it can coordinate the correct instances of those fragments of the split activity. Therefore, we use WS-Coordination's built-in extension mechanism to extend the *<wscoord:Register>* element so that it includes this information.

When a scope fragment begins, it sends to the registration service a registration message that also includes the address of the participant protocol handler for that particular scope instance. In return, the coordinator sends a registration response containing the address of the coordinator protocol handler that will receive all messages for that particular scope or loop instance from that particular participant.

The pieces of data identified above are used by the coordinator and the participants to generate the specific addresses for specific instances of each coordinator protocol instance and participant protocol instance. How they generate these addresses is implementation dependent: the WS-Coordination contract is just that if the exchanged addresses are used, the messages will be handled by the right protocol instance. We recommend using WS-Addressing [1] reference properties to encode information about which particular scope instance at hand. These properties can directly be the data items we have highlighted above (scope name, etc) or middleware generated ones (workflow engine created process instance ID, etc).

Traditionally, the identifier of a specific instance of coordinated work (i.e.: transaction id) is flowed in the coordination context after calling the activation service. For the problem specifically at hand in this work, using the activation service is superfluous so the instance identifier is placed in the registration message.

4. Join Failures

Recall that a split loop or scope may have a join condition. In BPEL, if a join condition evaluates to false, then the activity throws a fault known as 'join failure'. This fault may be suppressed using the attribute 'suppressJoinFailure="yes | no"'. Suppressing it is equivalent to surrounding the activity with a scope that has an empty fault handler for this particular fault. However, an implementation is not forced to create an actual scope, and in BPEL 2.0 [5] it was shown that if done in this manner there may be interference in the order of compensation (if present).

In this paper, we support splitting loops and scopes. Therefore, a split activity that will be skipped can only be skipped if it fires a join failure or it if is in a scope that is faulting. Handling join failures and faults properly are then all that is needed to ensure that a fragment does not hang if it is known that a sister fragment in another participant will never be executed.

As expressed in section 3.1, the join condition of the split activity is restricted to be the conjunction of the local join conditions at each fragment. In order to handle coordinating join failures on split loops scopes, we place messages in both protocols as follows: A participant whose fragment is ready to start without a join failure sends *starting*. On the other hand, a participant that encounters a join failure upon reaching a fragment of a scope or loop sends a *join failure* message to the coordinator. The coordinator waits for all fragments of the loop/scope to be ready (send starting/join failure), and if any had sent *join failure*, the coordinator sends a *complete after fault* message to all the fragments and the protocol ends in the 'dead path' state.

The restriction that the join condition is the conjunction of the local join conditions could be eased without loss of generality by providing the coordinator with the join condition definition, sending the status of the incoming links with the starting message from each fragment, and evaluating the condition at the coordinator side. However, this option was not chosen because although it is less restrictive, it obscures the behaviour of the fragments by placing an important piece of the business logic in the coordinator.

An alternative option for dealing with the join condition is to remove the messages dealing with join failure and the dead path state from the protocols, and instead modify the processes: inject a scope with an empty join failure handler around the split activity and change its 'suppressJoinFailure' attribute value to 'no'. The new scope itself will be split due to the rubber band effect but will never throw a join failure because it will have no links at its own boundaries. A local BPEL join failure fault, (thrown if the join

condition on a fragment evaluates to false) at the scope fragment level is thus treated as any other fault. It occurs in the parent scope, which must also be split by the rubber-band rule, and will be thrown and handled as any other. However, this forces an additional scope protocol instance for each split scope or loop. Instead, we manage to achieve the same behaviour by placing messages to deal with the join failure in the loops and scope protocols directly.

5. The Split Loop Protocol

We choose to have exactly one fragment be responsible for the loop condition. This can be specified in one of several ways: using an attribute on the split loop (isResponsible="yes|no"), making the condition itself opaque, or by referring to a 'magic variable', on each 'non responsible' fragment, that gets populated through a message from the coordinator with the value of the loop condition upon its evaluation. An implementation can choose to take any such approach for a split process.

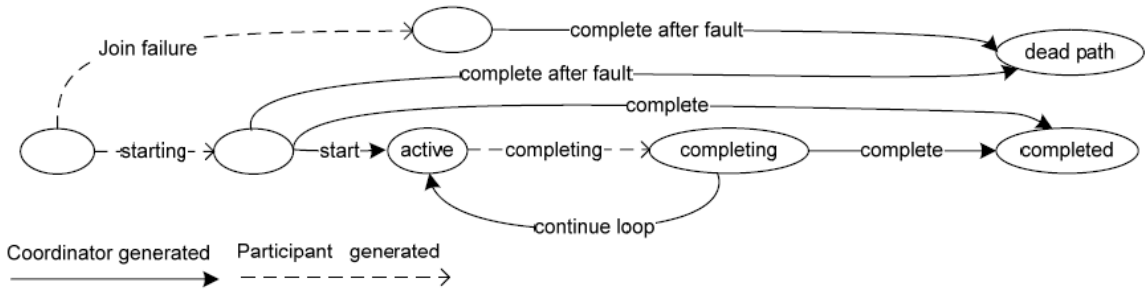

Figure 1: Participant Coordinator Messages, loop.

Figure 1 shows the behaviour of the protocol between a loop fragment and a coordinator for coordinating a split loop. Figure 2 shows the behaviour of the coordinator coordinating all fragments of that loop. Notice that any fragment can be the first to start, but the entire loop is only allowed to start once all fragments are ready. The same occurs for completion and iteration.

5.1 Participant-coordinator messages

The protocol takes place between each instance of a fragment of a loop and the coordinator. The coordinator gets a message that a loop is starting and containing the value of the while condition at that fragment. If the fragment is not the one responsible for the condition, then that value is omitted. The coordinator eventually sends back either a *complete* message if the loop condition is false or a *start* message so that the participant actually starts running its piece of the loop. Once the participant completes an iteration, it sends a *completing* message that again contains the value of the loop condition.

The coordinator then sends either *continue* if the loop condition is true or *complete* if the loop condition is false. The participant will then start a second iteration in the former case or navigate out of the loop in the latter case.

5.2 Participant behavior

The behaviour necessary on the participant side involves being able to listen, react to, and influence the process instance's behaviour. The participant side needs to be able to know when a loop fragment has started or had a join failure and to block the activity when this occurs. In needs to be able to unblock the fragment and let it either iterate or complete based on messages from the coordinator, regardless of the value of the local loop condition if the fragment is not responsible for the condition.

5.3 Coordinator behavior

Having explained the coordination messages between a fragment and the coordinator, we now look at the behaviour of the coordinator in coordinating all fragments of one loop for one instance of that loop.

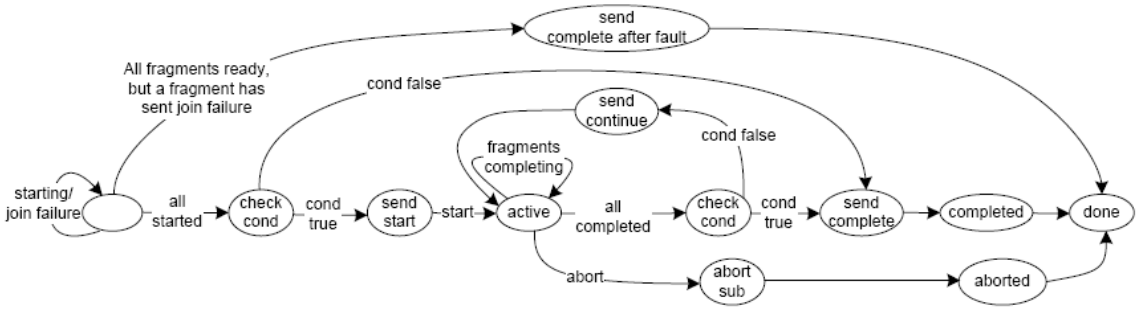

Figure 2: Coordinator Behavior, loop.

First, the coordinator waits for all instances of the loop to send *starting*, signalling that they have been reached in the control flow and are ready to run. The starting message contains the value of the loop condition from the responsible fragment. Based on this value, the coordinator then sends either *complete* or *start* to all fragments of the loop. Once start is received by a fragment, it starts running the body of the loop. Once it completes its work for one iteration, it sends *completing*. If it is the fragment responsible for the loop condition, then the *completing* message also contains the result of evaluating the condition. Then, the coordinator waits for all fragments to send *completing* and sends either *continue* if the loop condition is true or *complete* if it is false. If *continue* is sent, each fragment runs another iteration, otherwise it just completes and its process continues navigation out of the loop.

6. The Split Scope Protocol

In order to understand how to fragment a scope, one must understand how a BPEL scope behaves. In BPEL, a scope is a compound activity. It may be the source and/or target of control links. The behaviours that we are concerned of a scope are that it can define fault and compensation handlers for the activities nested within it. A fault handler consists of a part of a process that runs if a particular fault occurs. A compensation handler consists of a part of a process that provides an 'undo' action that can be called on a completed scope. The two are tightly related.

If an activity in a scope fails, all activities in that scope are terminated and a search for a fault handler is performed. If one is found on the parent scope of the faulting activity, then that handler runs next. If not, the search goes up the scope hierarchy until it does find a handler and execution continues there. A scope that faulted and did not have a corresponding handler fires all its outgoing links with a negative value. A scope that completed successfully or that successfully completed one of its fault handlers fires can then fire its outgoing links normally.

A fault handler may compensate one or more of the immediate children of the scope it is defined on. It may do so explicitly using the <compensate name="…"/> activity (renamed 'compensateScope' in BPEL 2.0). This activity may only appear inside a compensation handler or a fault handler.

In this paper, we handle compensation only for this explicit case in which the name of the scope to be compensated is provided and where the named scope must have an explicit compensation handler defined. The scope on which the compensation handler is defined may be split, the compensation handler itself may be split, and the scope in which the compensate activity is defined may be split (even if the named scope is not). Note that BPEL also defines default compensation that takes place due to either the <compensate/> activity (with no scope name in BPEL 1.1) or due to default fault and termination handling.

Default fault handling in BPEL is supposed to not only terminate the children but also compensate any completed children in 'default compensation order' and then rethrow the fault. Termination (also known as handling of the enclosing scope fault) in BPEL occurs in a running scope when its parent is exiting due a fault. The default behaviour is defined to trigger default compensation on any completed children.

Although the protocol shown in this paper was designed to handle default as well as explicit compensation, we believe the default aspect is too complex for the goals of this paper. Therefore, we will not at this time discuss the details of compensation that results from: default termination, default fault handling, and default compensation handling.

6.1 Data in split scope handlers

As mentioned earlier, this work focuses on sharing data that is necessary for the coordination of control and lifecycle for split loops and scopes.

Data provided with a fault in BPEL may be needed by a fault handler. For example, a 'throw' activity may have a variable attribute which requires that the value of that attribute reach the fault handler. Therefore, we send any data provided with the fault to the coordinator. The coordinator then sends it to all fragments that have a corresponding fault handler. A fault handler at a fragment that has a 'faultVariable' attribute will then save this message in this variable. For example, the value of variable 'x' is saved in variable 'y' if the fault was caused by a <throw faultName="ns:flt" variable="x"/> and caught by a <catch faultName="ns:flt" faultVariable="y">…</catch>.

As in BPEL 1.1, activities in a compensation handler can read data from variables in their own fragment based on the snapshot of the process state upon completion, and any data they write occurs on that snapshot and is not visible to any activities outside the handler. Again, sharing data between fragments, even for the compensation handler, is the responsibility of the splitting algorithm and not of the coordination work.

6.2 Effects on process instance creation

Special care must be taken in starting the fragments of the process itself (as opposed to those of loops and nested scopes). Unlike in the case of loops where one could simply wait for the other fragments to be started, the process fragments themselves may start very far apart in time (depending on when the message that can create an instance is received at each). For the case of process fragments that include protocol-driven split loops and scopes all fragments must be notified in case one of them fails, for both runtime (if another fragment has the necessary fault handler) and auditing purposes. Here, one cannot simply wait for all other process instances to start, especially since the start message of one fragment may come from another fragment.

The coordinator needs to start all process instances once it is known that at least one has started. This is enabled in this work by using a **'starter service'** that can create an instance of the process upon receiving a *startInstance*

message containing the correlation set value. In order to deal with a possible race condition, we place the requirement that the coordinator must ignore duplicate registration messages and the participant must ignore a *startInstance* if an instance with the same correlation set value already exists.

Starting an instance in this manner will result in all fragments of the process registering, starting, and being in the active state once one of them starts. It will not cause activities to occur out of order because in BPEL the first activities to occur are the 'create instance receives'. Receive is a blocking activity in BPEL, so these instance creating receives will block and wait for the actual external messages before allowing work in the body of the process to execute. Furthermore, since the split process was the result of splitting a larger BPEL process then starting all fragments at the same time is equivalent to having created one instance of the larger process.

6.3 Participant-coordinator messages

The messages between the coordinator and the participant for coordinating a fault are shown in Figure 3. Just as in the case of loops, one must also wait in this case for all fragments to start before allowing any particular one to actually begin executing its nested activities. The completion of the scopes is synchronized, also similar to the case of split loops. A BPEL scope, however, may complete either through having run its activities successfully, or through having run one of its fault handlers successfully. The two cases both lead to the scope having completed and its outgoing links being fired normally. However, in the latter case, the completion is considered abnormal and the engine needs to distinguish between the two especially if compensation is also involved. The result is that the fault handling protocol propagates a fault from one fragment to all other fragments of the scope that can handle it, and synchronizes the completion of the (split) fault handler.

For this protocol, all messages that start with *fault* include the fault QName and a fault variable value. All messages that are related to compensating a particular scope include the name of that scope.

Consider first the process fragments. Such a fragment will either start normally or be started by the coordinator. In the first case, it sends *starting*. In the second case, the coordinator sends a *startInstance* message containing the values of the correlation set to the fragment's 'starter service' as provided in the deployment information. This message is not part of the protocol, as is seen from the fact that it is not targeted at the participant service but at our new 'starter service'.

As each fragment creates its process instance, the process-level scope will send a registration message to the coordinator and everything continues as for any other (non-process) scope.

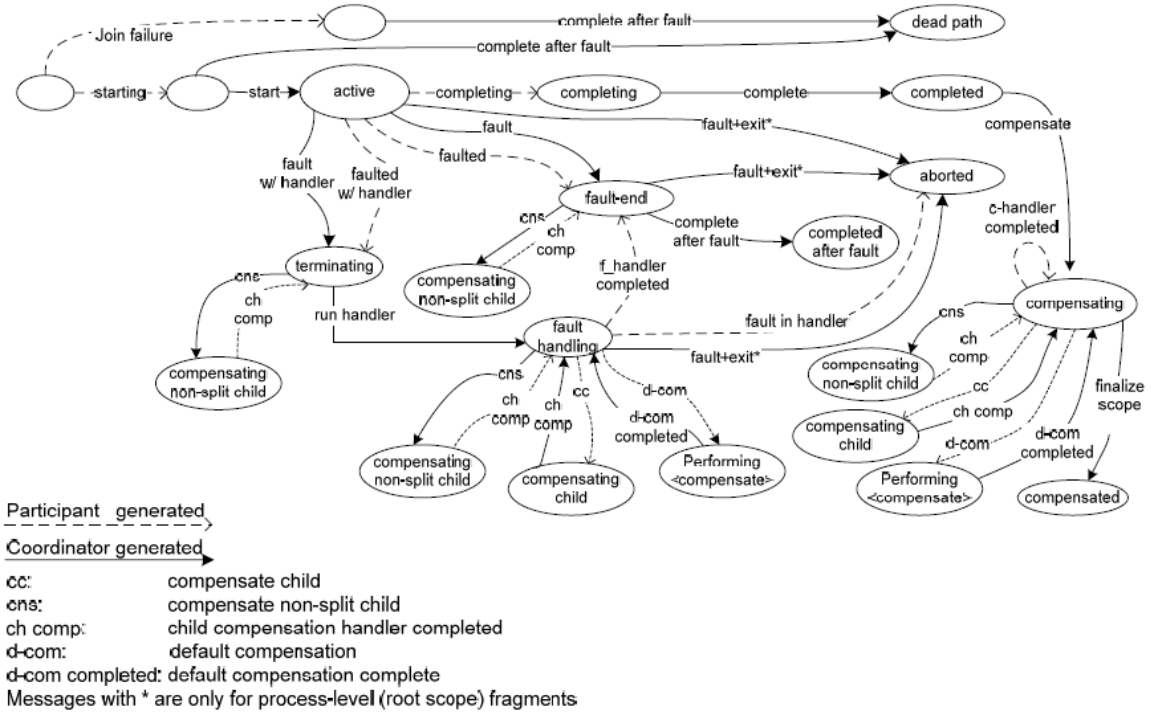

Figure 3: Participant Coordinator messages, scope.

A non-process scope sends *starting* when it is reached by navigation. After all fragments of a scope have started, the coordinator sends a *start* message to that fragment allowing it to actually start. A fragment stays in the active state while running its nested activities and no fault has occurred anywhere within the split scope. In the normal case, the next step would be to *complete* as was done in the case of loops (from active, to completing, to completed state).

In the case of a process-level scope fragment, the participant may get a *fault+exit* message from all states in which a fault may have been thrown: 'active', 'fault handling', and 'fault-end'. This occurs in case a fault has been encountered elsewhere and cannot be handled by any scope up to and including the level of the process itself. In other words, if the scope lookup algorithm in section 6.6 does not find a scope. This message cannot appear in the protocol of a scope that is not a fragment of the process-level scope.

Another step out of either the 'active' state occurs if another fragment faulted, and the scope of this fragment has a handler for the fault, then the coordinator would send a *fault w/ handler* message. If another fragment faulted and other fragments have handlers but not this one, then a *fault*

message is sent. If a fault is thrown and there are no handlers in any fragments of this scope, the coordinator will throw the fault to a parent scope's protocol. That is why the protocol does not deal directly with this case.

From the 'active' state the participant may also itself encounter fault. In this case, it sends a *faulted* message if it faulted and has no handler for the fault, or a *faulted w/ handler* message if it faulted and has a handler for that fault on the fragment itself. It then waits for the coordinator to confirm that it should run the handler (*run handler* message). As we are not dealing with default compensation at this time, we will ignore the other transitions out of the 'terminating' state.

After receiving either a *run handler* message, the participant runs the corresponding handler. The handler itself then either completes (*f_handler completed*) or faults. If the handler completes, then the coordinator waits for any other fragments of that handler to complete and then sends a *complete after fault* message.

Otherwise, if the handler faults, the participant sends a *fault in handler* message that is used by the coordinator to lookup an appropriate handler using the algorithm in section 6.6.

Next, we focus on compensation. The fault handler may have a <compensate scope="…"/> activity in one of its fragments. If this activity is reached, it sends the coordinator the request to perform this compensation work (*compensate child* or *compensate non-split child*). The target scope may be split, or may be non-split but in another fragment. The coordinator will request the compensation handler of the named scope and notify the participant once the compensation has completed (*child compensation handler completed*).

The scope to be compensated may be non-split and in a fragment that does not have part of the fault handler. This fragment will be in the 'fault-end' state when the request to compensate arrives at the coordinator. Therefore, it can accept requests to compensate non-split children while in this state.

Notice that the protocol also has a set of transitions and states that are present to enable the coordinator to request that a fragment run the compensation handler of a non-split child scope. These will only be the non-split child scopes that are relevant to the compensation order and therefore present in the scope tree. The protocol shows three states labelled 'compensating non-split child' that are there for this specific case. If the child to be compensated is not in the completed state, then the engine is to treat

this request as a no-op and send back immediately the *compensation handler completed* message.

A scope can only be compensated after it has completed, so the states for running the compensation handler of a split scope follow after the completed state. Split scopes can only be compensated through a command from the coordinator, as will be details in section 6.4. Therefore, the compensate message comes from the coordinator and sends the protocol from the state *completed* to the state *compensating*.

The coordinator sends *compensate* to all fragments of a split scope that it intends to compensate, whether that fragment has a handler or not. Any scope fragment with all or part of the compensation handler sends *compensation handler completed* upon completing the work in its part of the handler. Once the coordinator hears back from *all* fragments that have part of the compensation handler, it notifies all the scope's fragments that compensation has completed by sending the *finalize scope* message.

Finally, there are several states that are there to deal with default compensation.

6.3.1 Dealing with Race Conditions

Race conditions are introduced where one has a state with transitions that can be initiated either by the participant or by the coordinator. Looking closely at the protocol one can notice possible race conditions in the 'active' and 'fault handling' states.

For the 'active' state, there is a race between faults happening in different fragments, and a race between the participant wanting to complete and the coordinator trying to send it a fault. The former race already exists in BPEL, where faults may occur in parallel branches at nearly the same time. Only one fault is dealt with, the first one to reach the scope. Here, that will be the first one to reach the coordinator.

To ensure this, we place the rule that, for the 'active' state, the coordinator messages win over the participant messages. Therefore, if the coordinator has sent out a *fault, fault w/handler,* or *fault+exit* message, then *completing*, *faulted* or *faulted w/ handler* from the participant are ignored. The case of *completing* is especially relevant since the coordinator may in fact have received it but still decided to send a *fault* message because another fragment had faulted. Furthermore, notice that a fragment cannot start its handler even if the fault came from it and it has a handler. It needs to wait for the coordinator to send *run handler*. The reason is that if the participant could

immediately start its handler and there was in fact a race, then it could start a handler for a different fault than the coordinator has deemed the winner.

For a race between *handler completed* and *fault in handler* and the *fault+exit* messages, *fault+exit* wins. Therefore, the coordinator behaviour will ignore the *handler completed* or *fault in handler* message if it has already sent a *fault+exit*. Then, the race is not a problem because it only occurs for the process level scope with the fault+exit message. If this message comes late, then it can still be accepted by the protocol because it can also occur in the next state and also leads to 'aborted'.

Next, we address races between compensation related messages: those that lead to and from any of the states: compensating, compensating non-split child, compensating child, or performing compensate. These messages must not be dropped. Any of the compensation related messages that create a race must be queued and handled, unlike in the case of fault handling where one wins over the other. For example, if a fault handling fragment sends the coordinator a request to compensate a nested scope A and the coordinator sends it a request to compensate its nested non-split scope B, then fragment will queue the coordinator's request and handle it once it goes back to the state 'fault handling'.

In case of a race condition, fault related messages win over any compensation related messages.

6.4 Participant Behavior

Here as well, the participant side needs to be able to react to and affect the process instance's behaviour. It needs to be able to detect the scope fragment starting or encountering a join failure and then block it until it receives the *start* message. It needs to detect a fault and override local fault handling behaviour as will be described in the next section. It needs to be able to produce a fault in a process instance even if that fault did not originate in that fragment at all.

Finally, it needs to be able to completely disable default compensation handling on split scopes, whether triggered by fault or termination (enclosing scope fault) handling. Compensation for split scopes can only be triggered from the coordinator. The engine can only start an explicit compensation handler of a split scope and can only do so upon receiving a request from the coordinator.

This section is only concerned with non-split scopes that belong to the scope relationship tree. The default fault handler for the 'enclosing scope fault' (or

in BPEL 2.0, the default 'termination handler') of a non-split scope is left intact and without interference. Unhandled faults that arise from non-split scopes will reach a split scope and be handled as any other fault in the fragment.

The BPEL engine running a scope fragment may also only start the explicit compensation handler of a (relevant) non-split scope upon request from the coordinator. Both the engine (due to a compensate activity in the non-split scope's handler or due to termination) and the coordinator (if the non-split scope is to be compensated and has no explicit handler)are allowed to trigger the default compensation handler of these non-split scopes.

Recall that a request from the coordinator to compensate a non-split scope that has not completed must be treated by the engine as a no-op resulting in an immediate *compensation handler completed* message. Additionally, if the non-split scope is nested in a (split or non split) loop, the engine must determine whether it needs to compensate more than one instance of the non-split scope and the coordinator is not involved beyond triggering the compensation and being notified of its completion (all instances of a group, if applicable). In other words, the engine must treat a single request to compensate such a scope by name as if it had come from the instance itself.

6.5 Coordinator Behavior

Having described the fragment/coordinator messages, we now describe the corresponding behaviour of the coordinator when running an entire split scope; that is, when interacting with all of its fragments from start to finish. Figure 4 illustrates the behaviour of the coordinator.

The coordinator uses the relationship tree for identifying when all participants for a particular scope have registered, to search for handlers, and to relate endpoints to participant names. Words in italic correspond to signals of the protocol itself. Unless noted otherwise, a 'send' state sends to all fragments of this scope. Conditions on a transition are either local or they are signals received from participants (italics). The grey state and the underlined transition out of the active state are highlighted because they form the communication between several coordinator side scope behaviours. The protocol can be aborted at any time by a parent scope's protocol, hence the 'aborted' state on its own in the corner. Transitions were not drawn to it from every state as not to clutter the figure.

The states that deal with default compensation are the three states labelled 'compensation due to termination/rethrow' and the two states labelled 'default compensation on children'. Suffice it to say that for the latter two states, the compensation handlers of child scopes will be requested to run in

the order specified by a 'default compensation order graphs' that resides in the coordinator. For the former three, a complex interleaving of default compensation handling deep into the scope tree will take place due to the compensation order resulting from mixing compensation resulting from default termination with that resulting from default fault handling as the fault gets rethrown up the scope hierarchy.

Now, we take a closer look at the protocol. First, if a fragment is the process itself then one starts with the 'startInstance' message that will cause all fragments to create an instance and register if they have not already done so. Then, registration (not shown in the figure) takes place for the rest of the fragments and the protocol itself can begin. The behaviour for any scope is that the fragments send *starting* and the coordinator waits for all fragments to send this message before it sends *start*.

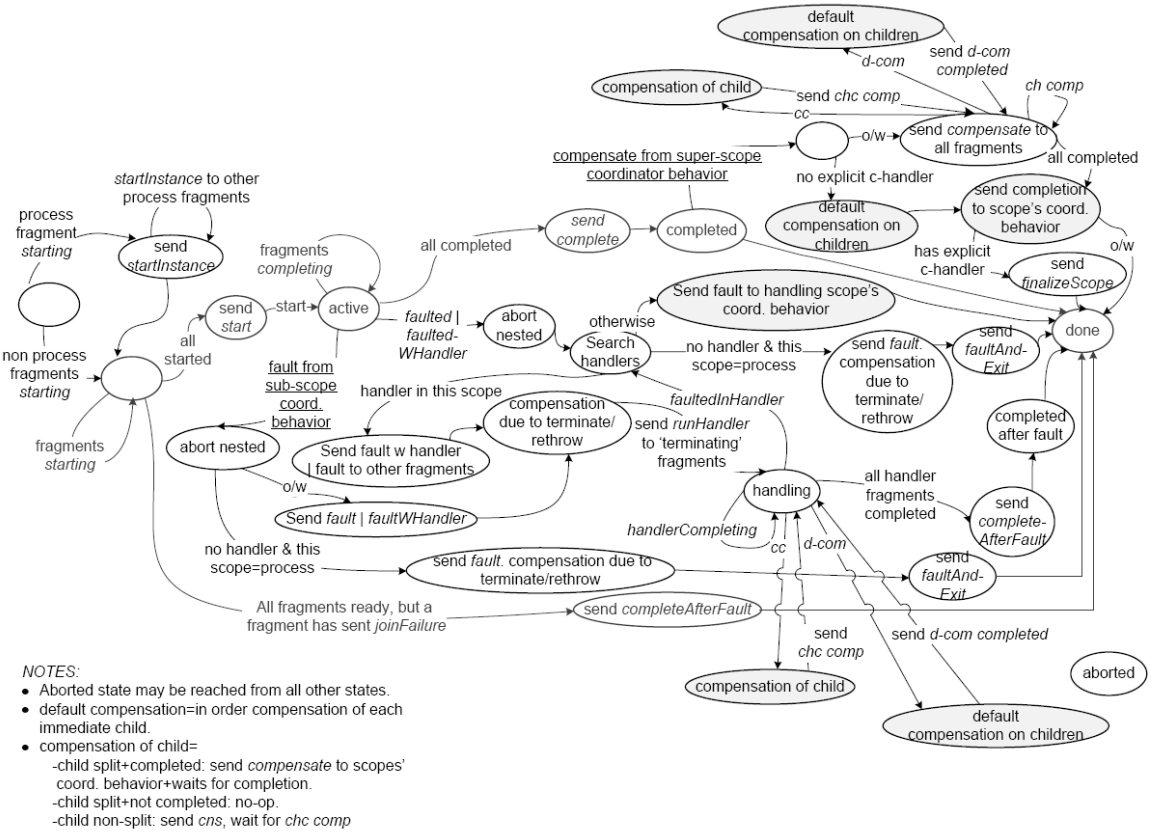

Figure 4: Coordinator Behavior, scope.

A fault from a fragment *(faulted/faulted w/ handler)* causes the coordinator to search the scope tree for a handler using the algorithm described in section 6.6. If the search does not yield a scope, then *fault* followed by *fault+exit* is sent to the fragments of the process level scope. Otherwise, if the handler is on the scope that faulted (in any of its fragments), then the same fault is

thrown by the coordinator to those fragments so they can handle the fault locally. If the message had been *faulted with handler*, then the coordinator sends *run handler* to that fragment, *fault* to any other fragments of the scope that have no handlers for the fault, and *fault w/ handler* to any other fragments that have a handler. As we are not dealing with default compensation, we will skip over the terminating state and so for this last case the coordinator will then also send *run handler*.

The previous chapter already mentioned how race conditions are dealt with in case of secondary faults from fragments.

If the handler is in a parent scope, then the coordinator will send a local fault signal (grey state) to the coordinator behaviour of the scope that has the handler. This signal appears in a scope's coordinator behaviour as the underlined transition out of the 'active' state. It will in turn abort all nested scope behaviours, and then send a message to the participants causing the fault to appear in all the fragments of the handling scope. At each fragment, such a fault will abort nested scopes as is normal BPEL fault behaviour. In this way, both the participant and coordinator sides of the nested protocols are stopped for all fragments without needing to send protocol *abort* messages to all. Default compensation may take place at this point as part of termination and of having caught the fault possibly several scopes higher than where it had been thrown. Next, the fault will kick off the fault handler fragments, and the protocol continues as shown.

If the fault cannot be handled by any scope, then *fault* followed by *fault+exit* are sent to the process-level fragment ending the instance in every process fragment.

The result is an interleaving of local BPEL behaviour and coordinated BPEL inspired behaviour (fault handler lookup, etc) to achieve the execution of cross-process fault handling scopes.

The search for handlers occurs in the coordinator using the knowledge of which scope the fault was caused in, the relationship tree, and the algorithm for finding a fault handler presented in section 6.6. Note that this means that the *fault* and *fault w/ handler* messages are only propagated from the coordinator to a scope that has a handler for the fault (regardless of which fragment(s) the handler(s) are in) or to the process itself.

Next we focus on compensation. A compensate request from a fragment can only occur while a scope is running a fault or compensation handler.

Consider the case that the scope to be compensated is not split. In this case, its parent must be split if the request to compensate has gone through the coordinator. The coordinator sends a message to request its compensation through its parent scope's protocol. As noted in the previous section, the participant side will send the *child compensation handler completed* message either if the scope is not in the completed state or once it has completed the handler otherwise.

Now consider the case that the scope to be compensated *is* split. If this scope has not completed successfully, the coordinator will immediately return that the compensation has finished without doing any work (no-op). If this scope has completed successfully and also has an explicitly defined compensation handler, then the coordinator will send a request to each fragment of that scope asking it to compensate. Once all fragments of the compensation handler have completed, the compensation due to the <compensate scope="..."> activity is considered completed.

If this scope has completed but has no explicit handler, then default compensation would occur (details are out of scope).

Scopes nested in loops may need to be compensated more than once. We handle this as follows. For every instance of a protocol of a split scope, the coordinator keeps one LIFO queue for every immediate split child scope that is not in the parents compensation or fault handlers and that has an explicit compensation handler. This queue contains the identifiers of all instances, in order of activation, of the child scope's protocol that ran after the protocol instance of the parent started and before the latter reached the completed or fault handling states. The protocol instances in each queue form what in BPEL 2.0 is known as a 'compensation instance group' for non-default compensation. Once a split child scope is requested to be compensated, the coordinator will run the compensation handler of *each* of these instances (if completed) based on the order specified by the queue. As a result, if the child scope had been nested in (one or more) loops, we would compensate it as many times as the loop(s) had run. On the other hand, if a non-split scope is nested in one ore more loops, then it is up to the process engine in which that scope is running to track and compensate multiple instances of the non-split scope if the coordinator requests that the scope be explicitly compensated. Note that the behavior of compensation instance groups for default compensation is more complex, requiring the coordinator to keep track of looping in the default compensation graph.

Finally, we also require that messages arriving at a scope whose protocol has been aborted at the coordinator side are ignored. This may occur if the message arrives between when the coordinator aborted it and when the

coordinator sends a fault to it or its parent scope causing the participant to locally abort as well.

6.6 Searching for the fault handler

Consider the relationship tree define in section 3.3.2. In this section, we present the algorithm used to detect the scope that will be responsible for handling a particular fault.

The coordinator is informed upon the occurrence of a fault and needs to look for a handler when the participant sends either a *faulted* or a *fault in handler* message.

Let the scope node in the scope relationship tree corresponding to the split scope that receives the fault message be s'_{rt_0}. Let s_{rt_0} be the BPEL scope where the search is to start, *n* be the fault name contained in the message from the participant, and *h* be a Boolean that is true if the message received was *fault in handler* and false if the message was *faulted*.

We use '⊥' to denote undefined. The root scope is the only scope for which there is no edge in the set of cp-edges whose first element is this scope. Consider s_r to denote the root scope. First, we define a function called `parent` that retrieves the immediate parent of a given scope using the scope tree:

$$parent(s) = \begin{cases} s' & \exists c \in C : c = (s, s') \\ \bot & \text{otherwise} \end{cases} \tag{10}$$

Notice that s_r is the only scope where $parent(s_r) = \bot$.

Recall that in BPEL, if a fault occurred in a fault handler and is not caught within that handler, then the search really starts in the parent of the scope in which the fault occurred. Therefore, s_{rt_0} is defined as:

$$s_{rt_0} = \begin{cases} s'_{rt_0} & \neg h \\ parent(s'_{rt_0}) & h \wedge s'_{rt_0} \neq s_r \\ \bot & \text{otherwise} \end{cases} \tag{11}$$

We define the following function, taking the name of a fault and a split scope node as input, to check whether a particular scope has a handler for the fault:

$$m(n, s) = \begin{cases} true & n \in \pi_2(s) \\ false & \text{otherwise} \end{cases} \tag{12}$$

Recall that one must skip the parent scope when a fault is crossing a fault handler boundary. In order to perform the skip, we check the Boolean in each scope node that specifies whether this scope is top level in a fault handler. The following function is then used to retrieve the scope that can handle the

fault. We use $\pi_i(x)$ to represent the i-th projection map on x. For example, $\pi_1((s_1, s_2)) = s_1$.

$$g(s) = \begin{cases} parent(s) & \neg\pi_3(s) \\ g(parent(s)) & \pi_3(s) \wedge s \neq s_r \\ \perp & otherwise \end{cases} \quad (13)$$

We define a function y that takes the name of the fault and the scope where the search must begin and returns either the fault handling scope or undefined if none is found. It is defined as follows:

$$y(n, s) = \begin{cases} s & m(n, s) \\ y(n, g(s)) & \neg m(n, s) \wedge s \neq s_r \\ \perp & otherwise \end{cases} \quad (14)$$

Finally, the scope that can handle the fault is found as follows. The message provides both h and n and is addressed to the protocol of s'_{rt_0}. At the coordinator, Equation 11 is used to retrieve s_{rt_0} from these values and the scope tree. Finally, the fault handling scope is retrieved using $y(n, s_{rt_0})$.

6.7 Relation to the Loop Protocol

Upon receiving a fault, a participant and the coordinator itself must abort all nested loop protocols. The nested loops are known from the relationship tree at the coordinator side. At the participant side, the nested loops are known from the process definition.

For non-default compensation, the loop protocol is related only in so far as that it enables the coordinator to keep track of compensation instance groups of split scopes nested in split loops, as described in section 6.5.

7. Conclusion

In this paper, we have shown how one can define split loops and scopes that encompass activities from multiple BPEL business processes. We highlighted the main problems like fragment identification, instance matching, and race conditions. We argued that using coordination is a natural way to solve this problem, and provided a particular solution using new WS-Coordination protocols for coordinating the fragments of such split scopes.

We limited the compensation handling explanation in this work to only the explicit case, leaving the details of default compensation handling for a separate paper. This is due to the complexity involved in the work that needs to be done on the coordinator side for calculating compensation order and performing the combined default fault and termination handling.

Acknowledgements

We gratefully acknowledge the input of: Oliver Kopp for a detailed review of a previous draft; Thomas Mikalsen and Dimka Karastayanova for feedback on early ideas; Michael Paluszek for working on an implementation of this approach.

References

[1] Box, D. and Curbera, F. (Eds). *Web Services Addressing (WS-Addressing)*. Published online at http://www.ibm.com/developerworks/library/ws-add, 2004

[2] Cabrera, L. F. et al (ed.) *Web Services Coordination (WS-Coordination)*. Online at ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf, Aug 2005.

[3] Curbera, F., Goland, Y., and Klein, J., and Leymann, F., Roller, D. and Thatte, S., Weerawarana, S. *Business Process Execution Language for Web Service v1.1*, Online at http://www.ibm.com/developerworks/library/ws-bpel, May 2003.

[4] Khalaf, R. and Leymann, F. *Role-based Decomposition of Business Processes using BPEL*, Proc. of ICWS 2006, Industry Track, Chicago, USA, September 2006.

[5] OASIS WSBPEL Technical Commitee, Web Services Business Process Execution Language Version 2.0, Committee Specification, January 2007. Published online at http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html