Web Services Flow Language (WSFL 1.0)

May 2001

By Prof. Dr. Frank Leymann, Distinguished Engineer
Member IBM Academy of Technology
IBM Software Group

Notices

The authors have utilized their professional expertise in preparing this report. However, neither International Business Machines Corporation nor the authors make any representation or warranties with respect to the contents of this report. Rather, this report is provided on an AS IS basis, without warranty, express or implied, INCLUDING A FULL DISCLAIMER OF THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgments

The Web Services Flow Language is the result of a team effort:

Francisco Curbera, Frank Leymann, Dieter Roller and Marc-Thomas Schmidt created the language and its underlying concepts.

Matthias Kloppmann and Frank Skrzypczak focused on its lifecycle aspects. Francis Parr worked on details of the example in the appendix.

Many others helped by reviewing and discussing earlier versions of the document, most notably Sanjiva Weerawarana and Claudia Zentner.

Note: IBM intends to work with partners on the creation of a standard in the subject area. This specification is the IBM input to the corresponding standardization effort.

Comments

Please send any feedback, technical or editorial, to Web Services at wbservcs@us.ibm.com or webservices/Raleigh/IBM@IBMUS.

Contents

1 Introduction

The Web Services Flow Language (WSFL) is an XML language for the description of Web Services compositions. WSFL considers two types of Web Services compositions:

- The first type specifies the appropriate *usage pattern* of a collection of Web Services, in such a way that the resulting composition describes how to achieve a particular business goal; typically, the result is a description of a *business process*.
- The second type specifies the *interaction pattern* of a collection of Web Services; in this case, the result is a description of the overall *partner interactions*.

1.1 Flow Models

In the first case, a composition is created by describing how to use the functionality provided by the collection of composed Web Services. This is also known as *flow composition*, *orchestration*, or *choreography* of Web Services. WSFL models these compositions as specifications of the execution sequence of the functionality provided by the composed Web Services. Execution orders are specified by defining the flow of control and data between Web Services. For this reason, in this document, we will also use the term *flow model* to refer to the first type of Web Services compositions. Flow models can especially be used to model business processes or workflows based on Web Services.

1.2 Global Models

In the second case, no specification of an execution sequence is provided. Instead, the composition provides a description of how the composed Web Services interact with each other. The interactions are modeled as links between endpoints of the Web Services' interfaces, each link corresponding to the interaction of one Web Service with an operation of another Web Service's interface. Because of the decentralized or distributed nature of these interactions, we will use the term *global model* in this document to refer to this type of Web Services composition.

1.3 Recursive Composition

WSFL provides extensive support for the *recursive composition* of services: In WSFL, every Web Service composition (a flow model as well as a global model) can itself become a new Web Service, and can thus be used as a component of new compositions. The ability to do recursive composition of Web Services provides scalability to the language and support for top-down progressive refinement design as well as for bottom-up aggregation. For these reasons, recursive composition has been a central requirement in the design of the WSFL language.

1.4 Hierarchical and Peer-to-Peer Interaction

WSFL compositions support a broad spectrum of interaction patterns between the partners participating in a business process. In particular, both hierarchical interactions and peer-to-peer interactions between partners are supported. Hierarchical interactions are often found in more stable, long-term relationships between partners, while peer-to-peer interactions reflect relationships that are often established dynamically on a per-instance basis.

1.5 Relation to Web Services Stack

The guiding principle behind WSFL is to fit naturally into the Web Services computing stack. It is layered on top of the Web Services Description Language (WSDL) [1]. WSDL describes the service endpoints where individual business operations can be accessed. WSFL uses WSDL for the description of service interfaces and their protocol bindings. WSFL also relies on an envisioned "endpoint description language" to describe non-operational characteristics of service endpoints, such as quality-of-service properties. Here, we will refer to this language as the "Web Services Endpoint Language" (WSEL), which is briefly introduced in Section 7 "Appendix A: Endpoint Properties Extensibility Elements." Together, WSDL, WSEL, and WSFL provide the core of the Web Services computing stack.

1.6 Document Organization

This document is organized as follows:

- Section 2 "Language Overview" provides an example and a brief overview of the WSFL language.
- Section 3 "Service Composition Metamodel" describes the metamodel underlying WSFL.
- Section 4 "Language Description" presents a detailed description of the elements of the language.
- Section 5 "Appendix A: WSFL Schema" features the schema for WSFL.
- Section 6 "Appendix B: Internal Activity Implementations" defines WSDL extensibility elements needed to bind to selective executables providing internal activity implementations.
- Section 7 "Appendix C: Endpoint Properties Extensibility Elements" features an initial set of extensibility elements that describe endpoint properties of activities.
- Section 8 Appendix D: "The Ticket-Order Example" contains main parts of a more complex example of WSFL usage.

2 Language Overview

Before getting into a more detailed description of WSFL, we will sketch two use cases for the application of Web Services composition.

2.1 Use Cases

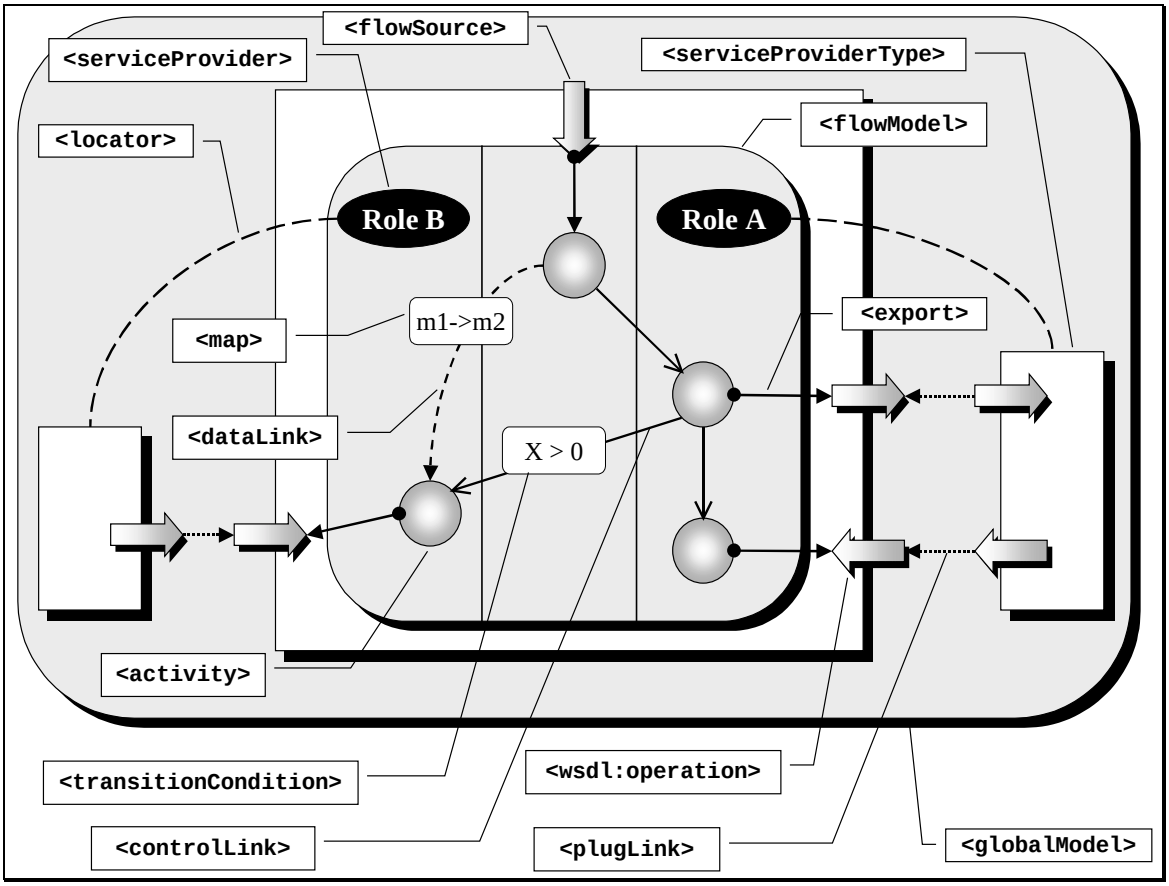
In the first use case, an enterprise wants to implement a business process for processing purchase orders using a set of Web Services.

They would identify the:

- B*usiness process* (for example, check credit history of the customer, reject order, process order, ship goods)
- *Business rules* for sequencing of these steps (for example, first check credit, then depending on the outcome, either reject the order or process the order followed by shipment of the goods)
- *Flow of information* between the process steps (for example, take purchase order as input to the process, pass it on to check credit, and so on).

In this "bottom-up" development scenario, they would find Web Services already offered by other vendors and companies that can be used to realize the various processing steps (for

example, a credit-checking service offered by a financial institution, a goods-production service offered by their favorite supplier, and a shipping service). They would then use WSFL to formally define the new business process.



A WSFL flow model defines the structure of the business process: WSFL activities (circles in the figure above) describe the processing steps, and WSFL data and control links represent the sequencing rules and information flows (eventually performing necessary data mapping) between these activities. For each activity, they would identify the WSFL service provider responsible for the execution of the process step (for example, services offered by shipping company A or by goods-supplier company B) and define the association between activities in the flow model and operations offered by the service provider using WSFL export and plug link elements. The resulting flow model is shown in the center of the figure above with ”swim lanes” representing the association of activities with service provider roles.

The second use case is a variant of the previous example. Here, an enterprise wants to offer a Web Service that mediates between service requesters (customers) who want to order goods and service providers who produce and deliver goods.

As in the previous example, the enterprise would define the business process for handling purchase orders as a WSFL flow model. In this case, however, they would not bind the activities to particular service providers. Instead they would identify the kind of service provider (role) they want for each activity (for example, some goods supplier for activity process order, some shipping service for activity ship goods).

They would then define the WSDL Web Service interface of the flow model, that is, the WSFL Service Provider Type of the flow model. This interface has two facets: One facet defines the interface that a customer would use when requesting processing of a purchase order, that is, the operations that the Web Service *provides* for use by service requesters. For example, the new service would provide an operation that takes a purchase order as input and passes

it on (through a WSFL flow source) to the activities in the flow model for processing. The other facet identifies the operations that the service *requires* from the other service providers.
For each activity, there is one (proxy) operation on the external interface of the flow model that the service would use to interact with a service provider implementing that activity. The resulting Web Service is depicted as the dark shape around the flow model in the figure above. This Web Service can now be advertised in a service repository where it would attract two kinds of parties: those who want to use services provided by the Web Service (in our case, customers who want to place orders) and those who want to play the role of a service provider (in our example, a shipping or a goods supplying service).

To make this model work, the activities in the flow model must be connected to operations that actually perform the process steps represented by each activity. This is done by a WSFL global model (the outermost box in the figure above), which describes the interaction between service providers and requesters. Our enterprise would use WSFL service provider locators to define criteria for selection of a particular service provider and WSFL plug links to associate operations on service provider elements with the service-requesting operations on the interface of the flow model.

2.2 A Quick Tour of WSFL

The purpose of a WSFL document is to define the composition of Web Services as a flow model or a global model. Both models have a declared public interface and an internal compositional structure. The composition assumes that the Web Services being composed support certain public interfaces, which can be specified as a *single port type* or as a collection of port types. We call this collection a *service provider type*.

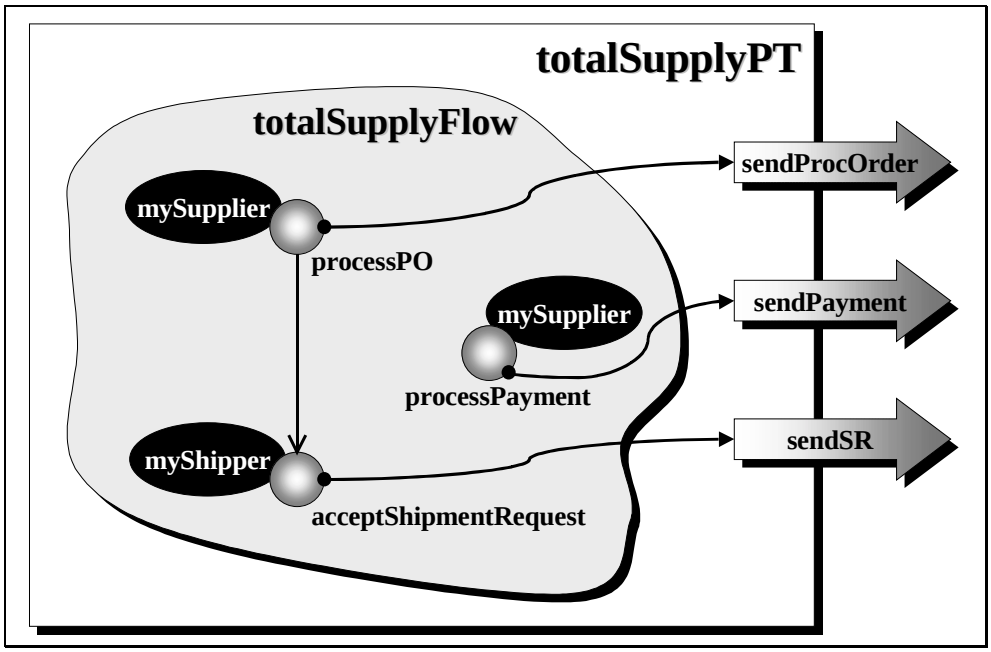
The following code is a simplified example of a WSFL service composition defining a flow model called `totalSupplyFlow`. The syntax of many elements has been abbreviated in the interest of conciseness. The example assumes a set of WSDL port type and operation definitions as public interface of the service provider types referred to: the `supplier` and `shipper` service provider types are somehow assumed by the flow model; the `totalSupply` service provider type appears to be defined by the flow model, but it has been already defined somewhere else, which is perfectly valid. Note that the flow model imposes “sequencing constraints” for the execution of operations of the `totalSupply` service provider type.

```
<flowModel name="totalSupplyFlow"
           serviceProviderType="totalSupply">

   <serviceProvider name="mySupplier" type="supplier">
      <locator type=”static” service=”qualitySupply.com”/>
   </serviceProvider>

   <serviceProvider name="myShipper" type="shipper">
      <locator type=”static” service=”worldShipper.com”/>
   </serviceProvider>

   <activity name=”processPO”>
      <performedBy serviceProvider=”mySupplier”/>
      <implement>
         <export>
            <target portType=”totalSupplyPT”
                    operation=”sendProcOrder”/>
         </export>
      </implement>
   </activity>
```

```
   <activity name=”acceptShipmentRequest”>
      <performedBy serviceProvider=”myShipper”/>
      <implement>
         <export>
            <target portType=”totalSupplyPT”
                    operation=”sendSR”/>
         </export>
      </implement>
   </activity>

   <activity name=”processPayment”>
      <performedBy serviceProvider=”mySupplier”/>
      <implement>
         <export>
            <target portType=”totalSupplyPT”
                    operation=”sendPayment”/>
         </export>
      </implement>
   </activity>

   <controlLink source="processPO" target="acceptShipmentRequest"/>

   <dataLink source="processPO" target="acceptShipmentRequest">
      <map sourceMessage="anINVandSR" targetMessage="anSR"/>
   </dataLink>

</flowModel>
```

The `totalSupplyFlow` flow model specifies how to collaborate with two service provider types in order to offer to their joint customers a complete business process. Each of the two service providers used within the flow model is represented by a separate `<serviceProvider>` element. One service provider is of type `supplier` and is referred to as `mySupplier` in the flow model. The other service provider is of type `shipper` and is called `myShipper`. Both service providers contain “binding” information as well. This information is provided by means of a `<locator>` element, which specifies the actual service that will be used when the model is instantiated. In this case, binding information is “static,” but more dynamic binding schemes are possible.

The business process represented by the `totalSupplyFlow` flow model consists of three business tasks, called *activities,* that have to be performed in order to successfully complete the business process: A purchase order has to be processed, a shipment request must be accepted, and money has to be received. Each of these activities is specified by a separate `<activity>` element.

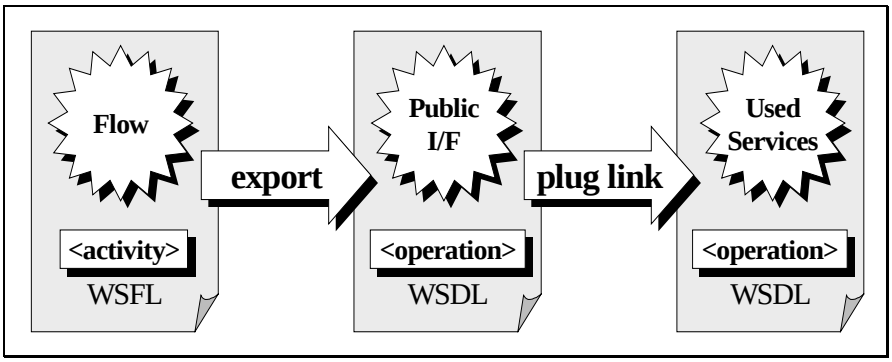


In our code example, the activities cannot be performed in any order, but there is a sequencing constraint between them: the processing of the purchase order by the supplier must precede the acceptance of the shipping request by the shipper; the money can be received at any time. The precedence rule is specified by simply connecting the two corresponding activities. Two kinds of connections are established, a control connection (through a `<controlLink>` element), and a data connection (through a `<dataLink>` element).

While the first connects the completion of one activity to the execution of another, the second connection represents a data exchange between the two. Note the `<map>` element nested inside the data link: it specifies what information needs to be transferred between the two linked activities. Also note that the separation of control flow and data flow is very helpful. For example, a service might only be enabled after the completion of another service without explicitly passing data from the former to the latter.

Web Services interact in a peer-to-peer manner. This pattern is immediately reflected by the interacting operations. For example, if a flow sends out a message via a notification operation, this operation corresponds to a one-way operation at a service provider. Pairs of corresponding operations in this sense are referred to as *dual* operations. In our example, the activity `processPO` has to send out a process order. For this purpose, the `totalSupply` service provider type declared by the flow model is assumed to include a port type `totalSupplyPT` with a `sendProcOrder` operation, which implements the activity. An `<implement>` element establishes this relation between an activity and its implementing operation. The service provider who is supposed to interact with an activity's implementation (for example, to process the message sent) is defined through a `<performedBy>` element.

To define the public interface of the composition, the `<flowModel>` element includes a declaration of the supported service provider type as an attribute of the flow model, and a mapping of operations of the port types of this service provider type to activities of the flow model. As indicated in the following figure, this mapping is specified by an `<export>` element, which relates an activity of the flow model and an operation of its public interface. This mapping defines the effect of each operation by relating it to the execution of the internal composition. The public interface defines the interaction of a flow model with the "outside," that is, it specifies which messages are sent and which are used.



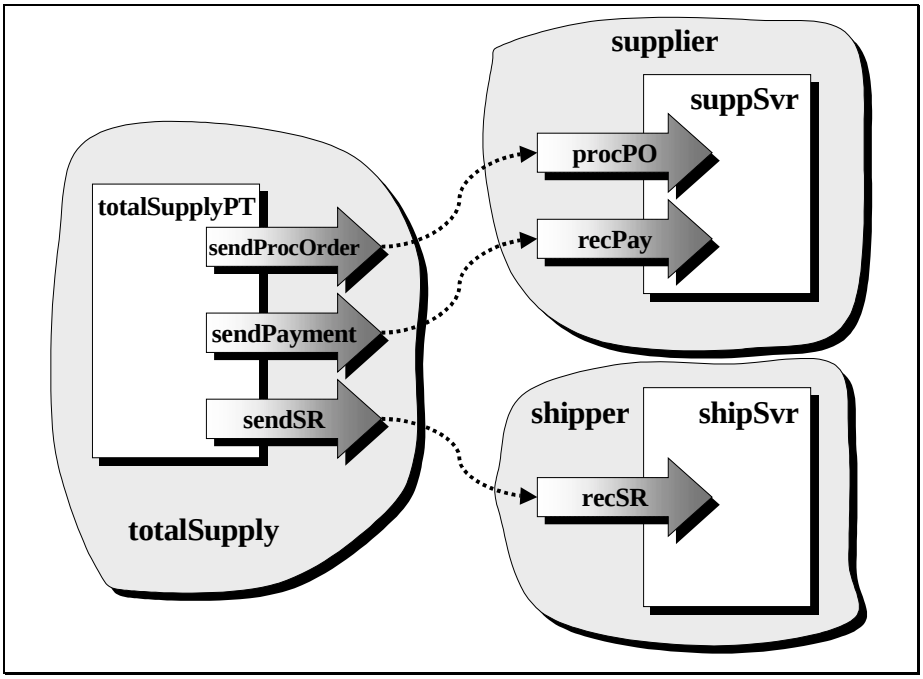
Typically, the operations of the public interface of the composition are not dual to any operation of the service providers to interact with, that is, messages are not simply sent to "anybody" or accepted by "anybody." Messages are related to a particular operation of a particular port type of the performing service provider. As indicated in the figure, the relation between an operation of the public interface of a flow model and an operation of a service provider is established through a `<plugLink>` element. Thus, a plug link represents the inherent client/server structure of a Web Service.

In WSFL, plug links are typically specified within a `<globalModel>` element (although plug links can be specified "inline" within a flow model). Note that the advantage of separating plug links from flow models is that relations between operations of arbitrary port types or service provider types can be defined, whether they stem from a flow model or not. From a flow model perspective, a global model makes the interactions between service providers explicit. The following example specifies the interactions between a supplier, a shipper, and a total supplier.

```
<globalModel name="mySupplyChain"
             serviceProviderType="supplyChain">

   <serviceProvider name="mySupplier" type="supplier"/>
   <serviceProvider name="myShipper" type="shipper"/>
   <serviceProvider name="myTotalSupply" type="totalSupply">
      <export>
         <source portType="supplyLifeycle" operation="spawn"/>
         <target portType="manageChain" operation="order"/>
      </export>
   </serviceProvider>

   <plugLink>
      <source serviceProvider="myTotalSupply"
              portType="totalSupplyPT"
              operation="sendProcOrder"/>
      <target serviceProvider="mySupplier"
              portType="suppSvr"
              operation="procPO"/>
   </plugLink>

   <plugLink>
      <source serviceProvider="myTotalSupply"
              portType="totalSupplyPT"
              operation="sendPayment"/>
      <target serviceProvider="mySupplier"
              portType="suppSvr"
              operation="recPay"/>
   </plugLink>

   <plugLink>
```

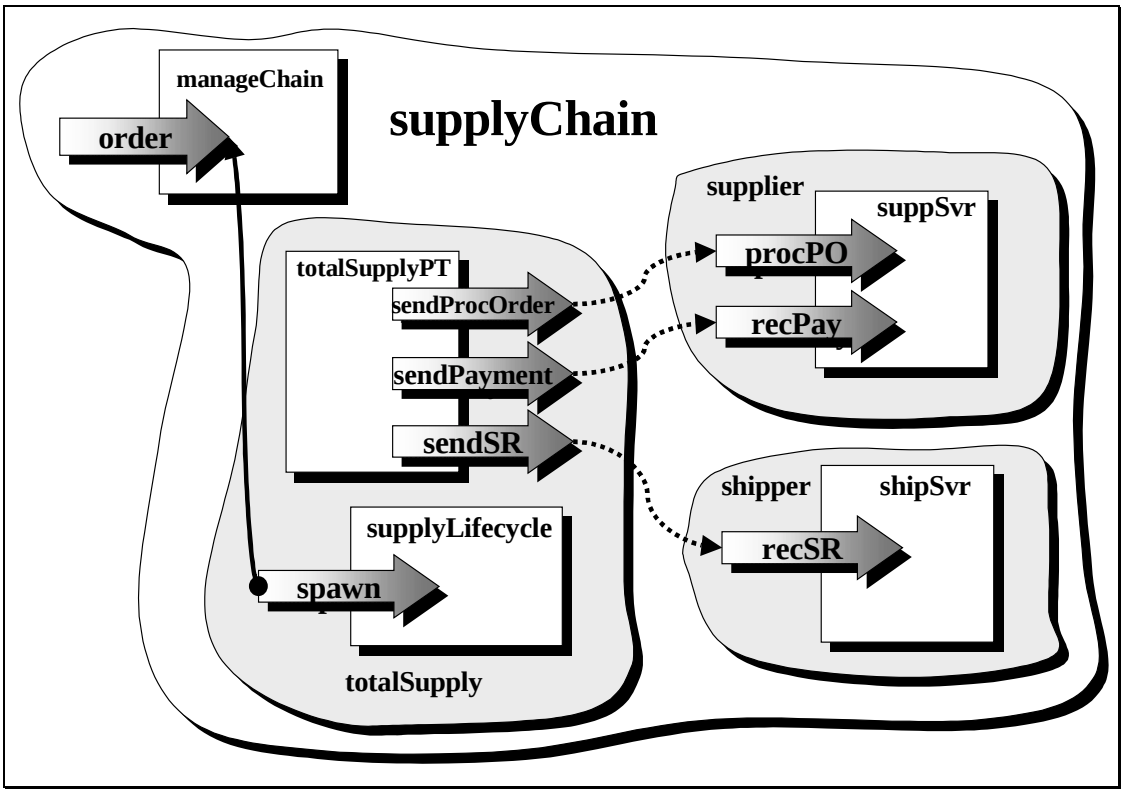
```
      <source serviceProvider="myTotalSupply"
              portType="totalSupplyPT"
              operation="sendSR"/>
      <target serviceProvider="myShipper"
              portType="shipSvr"
              operation="recSR"/>
   </plugLink>

</globalModel>
```

In the example, the `supplier` service provider type is assumed to support the port type `suppSrv` with two operations: one for processing a purchase order, and one for receiving a payment (`processPO` and `recPay`). The `supplier` service provider type may also define restrictions on the sequencing of the two operations (for example, the execution of the first operation must precede the execution of the second). For this purpose, the service provider type could be defined as the public interface of another flow model (but this is not done in our example). The `shipper` service provider type is assumed to support the `shipSrv` port type including an operation for accepting and processing shipping requests (`recSR`). The `mySupplyChain` global model now plug links the operations of these two service provider types with the `totalSupply` service provider type declared by the `totalSupplyFlow` flow model. The following figure depicts the plug links established by the example as dashed arrows.



As each composition, the `mySupplyChain` global model of the example declares a service provider type named `supplyChain`. This is done through an attribute of the `<globalModel>` element. The service provider `myTotalSupply` exports the operation `spawn` from its port type `supplyLifecycle` to the operation `order` of the port type `manageChain` that represents the public interface of the sample global model. The `spawn` operation is a lifecycle operation that allows the flow to kick off an instance of the `totalSupplyFlow` flow model: Thus, invoking the `order` operation, which is delegated to the `spawn` operation, will kick off the flow and will finally result in making use of the specified plug links. The first plug link specifies that the “sendProcOrder” operation of the public interface of the flow sends a message to the `procPO` operation of the `suppSvr` port type of the supplier. The other plug links are similar.

The following figure depicts the complete `supplyChain` service provider type defined by the `mySupplyChain` global model.



3 Service Composition Metamodel

This section describes at the conceptual level how Web Services are wired together into flows that represent business processes (see Section 3.1 “Flow Metamodel”). Section 3.2 “Lifecycle Interface” describes how instances of such a business process are manipulated as a whole. In Sections 3.3 “Business Process Lifecycle” and 3.4 “Activity Lifecycle,” we sketch a minimum set of states and the transitions between them that further describe a business process and each of its encompassed activities. Finally, Section 3.5 “Recursive Composition Metamodel” gives an overview on how new Web Services are composed out of other Web Services.

3.1 Flow Metamodel

This section describes the main concepts of the metamodel underlying WSFL for specifying flows. This is done by describing its syntax as a special kind of directed graph (Section 3.1.1 “Activities”) and its semantics by showing how each of the syntax elements is to be interpreted in concert with the other syntax element (see Section 3.1.2 “Operational Semantics”).

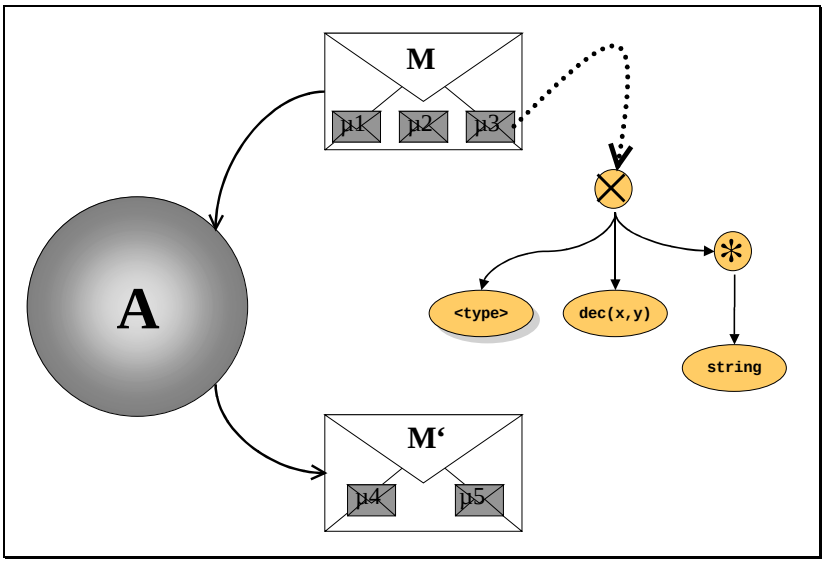
3.1.1 Syntax

This section describes the various ingredients of the metamodel in detail and explains their operational semantics.

3.1.1.1 Activities

Operations of Web Services are used within business processes as implementations of activities. An *activity* represents a business task to be performed as a single step within the context of a business process contributing to the overall business goal to be achieved. The operation used may be perceived as the concrete implementation of the abstract activity to be performed. Refer to Section 3.5.4 “Which Operation Is the Activity Implementation?” for more details.

Activities correspond to nodes in a graph. Each activity has a signature that is related to the signature of the operation that is used as the implementation of the activity. Thus, an activity can have an input message, an output message, and multiple fault messages. Each message can have multiple parts, and each part is further defined in some type system.



The figure above depicts an activity A with input message M and output message M'. Input message M has three message parts called μ_1, μ_2, and μ_3. Output message M' has two message parts, called μ_4 and μ_5. Message part μ_3 is defined through an XML schema the root of which is a <sequence> that contains some other complex type, a decimal simple type and a simple type that may hold multiple string fields.

3.1.1.2 Control Links

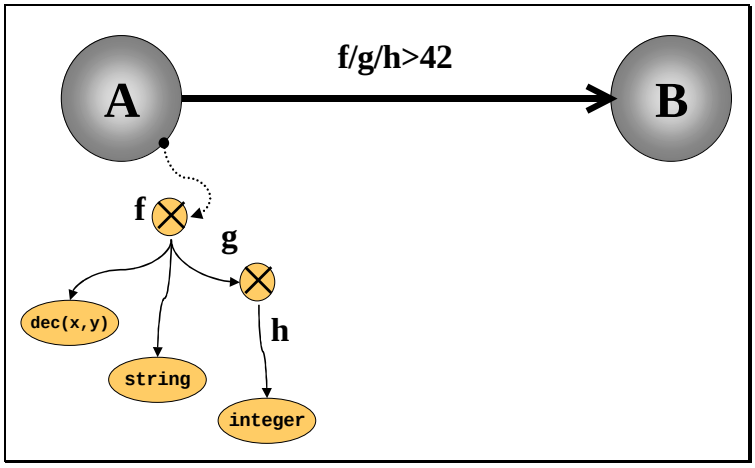
Activities are wired together through *control links*. A control link is a directed edge that prescribes the order in which activities will have to be performed (that is, the potential "control flow" between the activities of the business process). The endpoints of the set of all control links that leave a given activity A represent the possible follow-on activities A_1, …, A_n of activity A.

3.1.1.3 Transition Conditions

Which of the activities A_1,…, A_n actually have to be performed in the concrete instance of the business process (that is, the concrete business context or business situation) is determined by so-called *transition conditions*. A transition condition is a Boolean expression that is associated with a control link. The formal parameters of this expression can refer to messages that have been produced by some of the activities that preceded the source of the control link in the flow.

When an activity A completes, exactly those control links originating at A are followed to their endpoints the transition conditions of which evaluate to true. This set of activities is referred to as "actual follow-on activities" of A in contrast to the full set {A_1,…, A_n} of "possible follow-on activities." It is said that "control flows from A to the actual successors of A," or that the "control flow visits the actual successors of A," or that "navigation proceeds from A to its actual successors," or something similar like that.

In the following figure, activity B might need to be performed after activity A completes. The transition condition of the corresponding control link is specified as an XPath expression that references the output message of A: Activity B will be performed ("control flows to B" or "navigation proceeds to B") if, and only if, the integer value returned by A will have a value greater than 42.



3.1.1.4 The Origin of Flow Dynamics

Note especially that this mechanism is the origin of the whole dynamics within the control flow of business processes: Activities produce actual data values for their output messages, and these values will be substituted as actual parameters of the formal parameters of transition conditions. Exactly those control links will be followed whose transition conditions evaluate to true in their actual parameters. And exactly the endpoints of those control links

are the activities that have to be performed next "in the current *business context*." Thus, whenever an activity completes, that is, the operation of the Web Service that implements the activity returns data, this actual data can be made the basis for deciding which activities have to be performed next. And these activities are typically highly dependent on the data returned.

3.1.1.5 Control Links As Edges

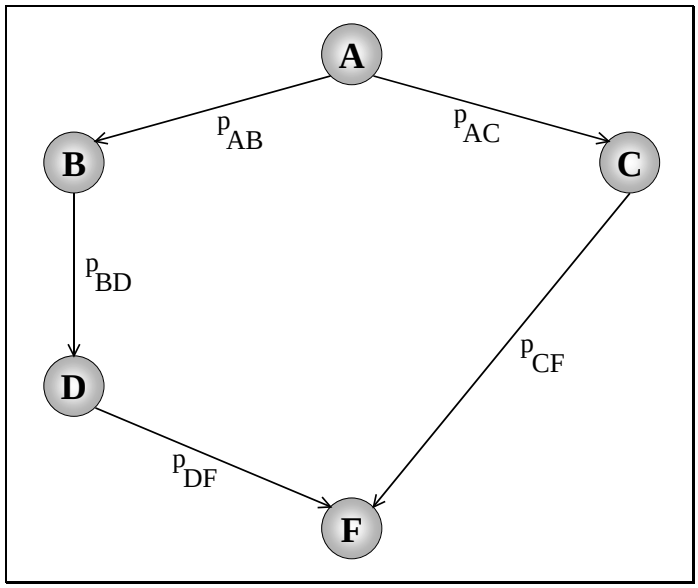
Control links are the first kind of edges in the graph structure that we use to represent models of business processes, or simply, flows. First of all, such an edge is directed, pointing from its source activity to its target activity, that is, from an activity to its (or one of its) potential successor activities. Next, such an edge is "weighted" by a transition condition, determining the actual flow of control. We do allow at most one control link between two different activities. Finally, the resulting directed graph must be acyclic, that is, we do not allow loops within the control structure of a flow (however, see Section 3.1.1.11 "Loops," on how loops are supported in a controlled manner).

Note that tools supporting the graphical construction of WSFL-compliant flow models can choose to support drawing loops. But the loops supported by the tool must be able to be transformed into the restricted variant of loops supported by WSFL. This restricted variant basically corresponds to "do until" loops.

3.1.1.6 Forks And Parallelism

An activity (like activity A in the following figure) is called a *fork* activity if it has more than one outgoing control link. When activity A completes, all control links leaving A will be determined and all associated transition conditions (p_{AB} and p_{AC} in the figure) will be evaluated in their actual parameters. The target activities of all control links whose transition conditions evaluated to true are exactly the activities that are to be performed next within the flow. For example, if p_{AB} evaluated to true but p_{AC} evaluated to false, exactly activity B will be scheduled to be performed; if p_{AB} evaluated to false and p_{AC} evaluated to true, exactly C is to be performed next.

In case both p_{AB} and p_{AC} get the truth-value of true assigned based on the actual parameters, and both activities B and C will have to be performed next. (We will explain later what happens along paths that are determined by a control link whose transition condition evaluated to false. See 3.1.2.1 "Death-Path Elimination"). In particular, it is very easy to achieve parallelism in the execution of flows: Simply introduce a fork activity and the "subgraphs" that are spawned-off by the control links with a true transition condition will be performed in parallel.



3.1.1.7 Joins and Synchronization

Typically, parallel work has to be synchronized at a later time. Synchronization is done through join activities. An activity is called a *join* activity (like activity F in the figure above) if it has more than one incoming control link. By default, the decision whether a join activity is to be performed or not is deferred until all parallel work that can finally reach the join activity has actually reached it (see 3.1.1.8 “Join Conditions” for potential deviations from this default behavior). In the figure above, when p_{AB} and p_{AC} had been evaluated to true, B and D can be performed in parallel with C, and F cannot be performed until control passed from C to F and from D to F. At that time, the truth-value of the transition conditions p_{DF} and p_{CF} are known; based on these truth-values it can be specified whether F should be performed if, and only if, both parallel executions successfully reached F (“p_{DF} `AND` p_{CF}”), or whether it suffice that at least one of the parallel executions reached F successfully (“p_{DF} `OR` p_{CF}”), and so on.

3.1.1.8 Join Conditions

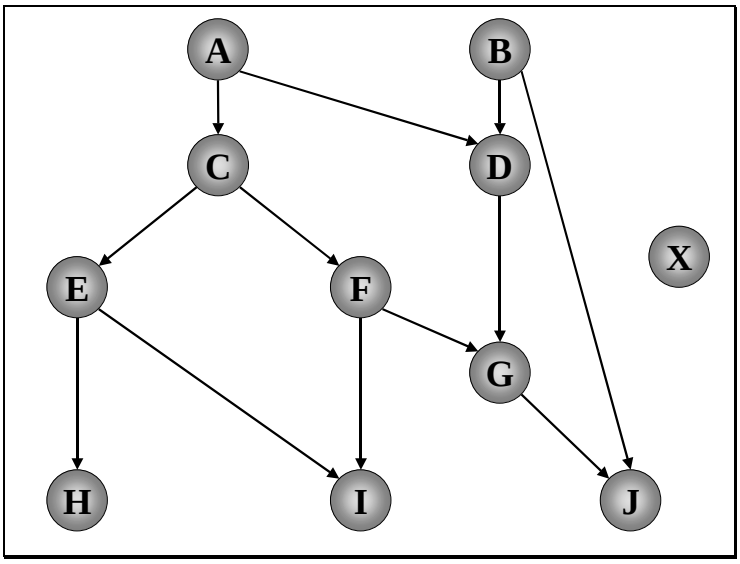
Thus, the truth-values of transition conditions of control links that enter a join activity allow for a more fine-grained mechanism of synchronizing parallel work at join activities. This mechanism is introduced through join conditions: A *join condition* is a Boolean expression associated with a join activity, and the formal parameters of this expression refer to the transition conditions of the incoming control links of the join activity.

Work along parallel paths reaches a join activity at different points in time. For example, activity C in the figure before might have been completed fast and the transition condition p_{CF} is evaluated while B is still running, that is, the transition condition p_{DF} gets evaluated at a later point in time. By default, the decision whether F is to be performed or not is deferred until p_{DF} has also been evaluated, even if the join condition is “p_{DF} `or` p_{CF},” for example, and is known to be true long before the truth-value of p_{DF} is known.

Thus, join conditions are really a means to synchronize parallel work, that is, to wait until parallel work comes to an end and then decisions can be made how to proceed. Sometimes, a weaker semantics of synchronization is appropriate and supported by the metamodel of WSFL: As soon as the truth-value of a join condition is known, the associated join activity is dealt with accordingly (that is, either performed or skipped). Control flow that reaches the corresponding join activity at a later time is simply ignored.

3.1.1.9 Start and End Activities

But what about activities that have no incoming control connector (like A, B, and X in the following figure), or outgoing control connector at all (like H, I, J, and X)? These kinds of activities are called *start activities* or *end activities*, respectively. In the following figure, activities A, B, and X are start activities, and activities H, I, J, and X are end activities.



Conceptually, each activity has a join condition associated: A node with a single incoming control link can be perceived as having a join condition that consists of the transition condition of the incoming control link. A start activity can be perceived as having a trivial join condition that consists of the constant "true" predicate. With this convention in mind, an activity can be started whenever its join condition is fulfilled. In particular, the join condition of an activity with no incoming control link is fulfilled when the flow model is "started," thus, the corresponding activities are "start activities" also from that perspective.

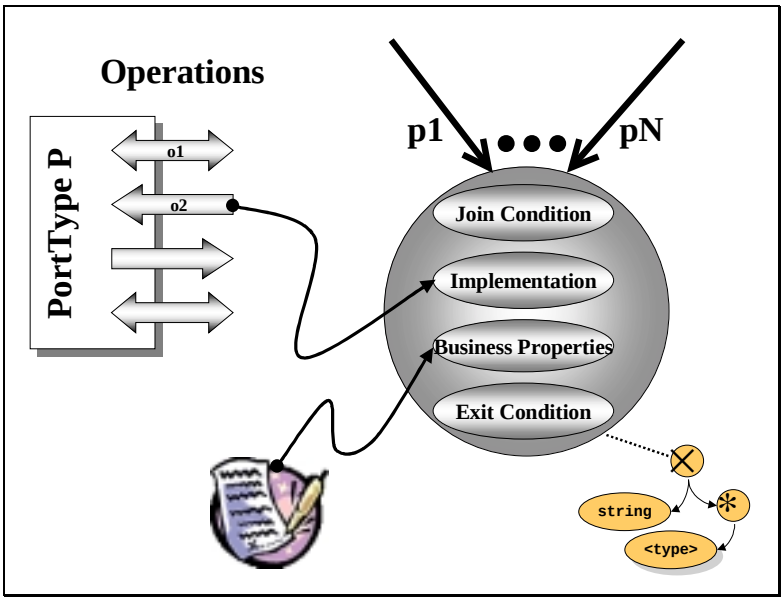
When a flow model is instantiated, all of its start activities are determined and scheduled to be performed. Based on the start activities of a flow, the "regular" navigation through the graph representing the flow model continues. That means, when a start activity completes, its actual successors are determined based on the control links originating at the completed start activity.

When an end activity completes, navigation stops at this point because there is no possible follow-on activity and thus, no actual successor to determine. But navigation might continue in other parts of the graph, thus, a lot of activities of the overall flow might still be awaiting their execution. But if all end activities within the graph have been reached, the overall flow is done. When the last end activity completes, the output of the overall flow is determined and returned to its invoker; and then, the flow ceases to exist.

3.1.1.10 Exit Conditions

The following figure summarizes the flow-relevant fine structure of an activity introduced so far. An activity is linked to the operation of a port type as its implementation, and if the activity is a join activity, it has an associated join condition. What is also shown is the exit condition associated with an activity: An *exit condition* is a Boolean expression, the purpose of which is to determine whether or not the execution of the implementation of the activity completed the business task represented by the activity. The expression can refer to the output message of its associated activity or even to output of any activity that ran before on the control path of the subject activity; the expression of an exit condition is provided in XPath syntax like the expression of a transition condition is.

The exit condition is evaluated once the operation of the implementing port type terminates. If the exit condition evaluates to true, the activity is treated as "completed." If the activity is completed, navigation continues and the next activities to be performed are determined based on the just-completed activity; otherwise, the activity is executed again.



For example, the exit condition can check particular reason codes or return codes of the activity implementation In doing so, the activity can be retried if a code indicates an implementation problem (for example, “automatic rollback due to detected deadlock”). Or the application already aggregates lower-level reason codes and provides a return code that basically says whether the implementation executed correctly or not. Or the exit condition checks a field that is implicitly set by a user (“The customer did not answer the phone call–I’ll try at a later time”). As all of these examples show, the exit condition allows to distinguish two events, namely the event that signals that the activity implementation returned from the event that signals that the associated piece of work (the business task) completed successfully. And navigation typically should continue only if the business task completed and not if the implementation has been interrupted for whatever reason.

3.1.1.11 Loops

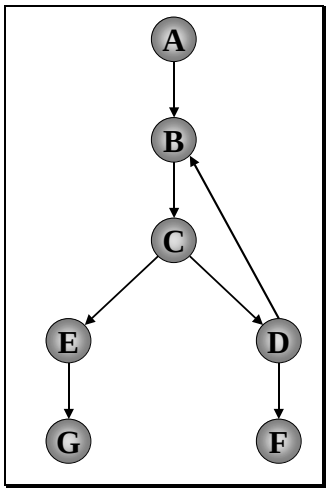
But there is another important use of exit conditions, namely for looping: An activity is iterated until its exit condition is met. Often, this mechanism for realizing do-until loops is used when an activity is implemented by another flow, that is, by means of the `call` lifecycle operation (see Section 3.3 “Business Process Lifecycle” and 4.6.5.2 “Lifecycle Operation `call`”). Because the metamodel does not support cyclic graphs, cycles must be realized by separate flows that are iterated based on exit conditions. This enforces a block-oriented specification of loops well known from structured programming.

Supporting arbitrary loops would allow specifying situations that are ambiguous, difficult to model unambiguously, and much more difficult to comprehend. The following figure shows a cyclic graph. Assume that control flows from A to B to C, and D and E are actually executed. We further assume, that when D completes, navigation can proceed to B again. When B completes the second time, control flows to C, and may continue to E and D again. Many problems and questions come up, for example:

- B is a join node. When control flows from A to B (the first time) the truth-value of the transition condition of the control link from D to B is unknown. The join condition of B must be an expression in ternary logic to specify the appropriate behavior.

- When C completes the second time, should control really flow to E again? Or does the intended loop just consist of B, C, and D? If the control flow should proceed to E, it might happen that E is still running because of its first invocation. What should happen in this situation? Should E be immediately interrupted and started again, or should the completion of E be awaited before its next invocation?

- When D completes and control flows back to B, and could also flow to F, should F be really started? Or should only the "backward control link" be honored? If F should be started, the same questions occur as for E before.



3.1.1.12 Data Links

There is a second kind of directed edges in the graphs of the metamodel, the so-called data links. A *data link* specifies that its source activity passes data to the flow engine, which in turn has to pass (some of) this data to the target activity of the data link. For example, the next figure depicts that activity A expects input data from activity B, which is indicated by a dashed directed edge (while we use solid edges to draw control links).

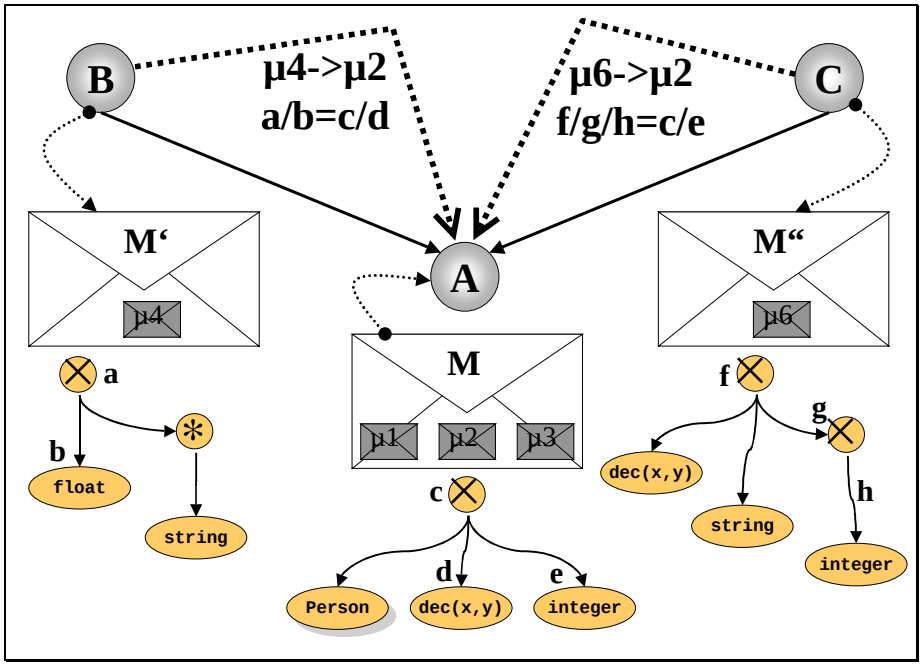
To make this meaningful, a data link can be specified only if the target of the data link is reachable from the source of the data link through a path of (directed) control links. Thus, "data always flows along control links." This makes sure in an easy manner that a couple of error-prone situations are avoided. For example, the spectrum of such situations extends from trying to consume data that has not been produced yet, to dead-lock situations in which one activity requires data from another activity as input but the latter activity needs the output of the former as its own input.

It is not required that data be always passed to an immediate successor of its producer. Many different activities might be visited along the path made from control links from the source of a data link to the target of the data link.

An activity might be the target of multiple data links. For example, this allows aggregating input from multiple sources, or it allows specifying alternative input from activities from alternative parallel paths. To facilitate this, data links are weighted by so-called map specifications. A *map* prescribes how a field in a message part of the target's input message of a data link is constructed from a field in the output message's message part of the source of the data link. It even allows that multiple maps to be defined for the same message part target. This is needed, for example, when alternative paths in the control are specified and data needed further on can be produced along each of the paths. If more than one map can be applied at run time, precedence rules specify which map to apply (see Section 4.5.1.4 "Container Materialization").

For example, activity A in the following figure expects input from both activity B and activity C. This is specified through two data links, one pointing from B to A, and the other pointing from C to A. The input message M of activity A consists of three message parts, called μ_1, μ_2, and μ_3. Message part μ_2 is described by a root element **c** that consists of a <sequence> of elements amongst which a simple decimal type **d** and a simple integer type **e** is found. The output message M' of activity B consists of message part μ_4 which is modeled as a <sequence> called **a,** which in particular, includes a simple float type **b**. The output message M" of activity C includes a message part μ_6, which is a <sequence> called **f** that contains a data element **g**, which in turn, is another <sequence> that encompasses a simple integer type **h**.

A map associated with the data link from B to A specifies that message part μ_4 of message M' contains a field that is to be transformed into a field of message part μ_2 of message M; furthermore, the target field in μ_2 is **c/d** (XPath notation) and the source field in μ_4 is **a/b** (if needed, an additional conversion routine can be defined). Similarly, a map is associated with the data link from C to B, and this map specifies that field **f/g/h** of message part μ_6 of message M" has to be copied to field **c/e** of message part μ_2 of message M.



3.1.1.13 Input and Output of Flows

Flows themselves can have input as well as output. To pass data from the input of a flow to the input of (some of) its encompassed activities, data links and maps can be used to. For this purpose, a *flow source* element is provided as part of the definition of a flow model that can be used as the source of data links that target at activities within the flow. Similarly, a *flow sink* element can be the target of data links that originate at activities within the flow. The flow source represents the input message of the flow, and the flow sink represents its output.

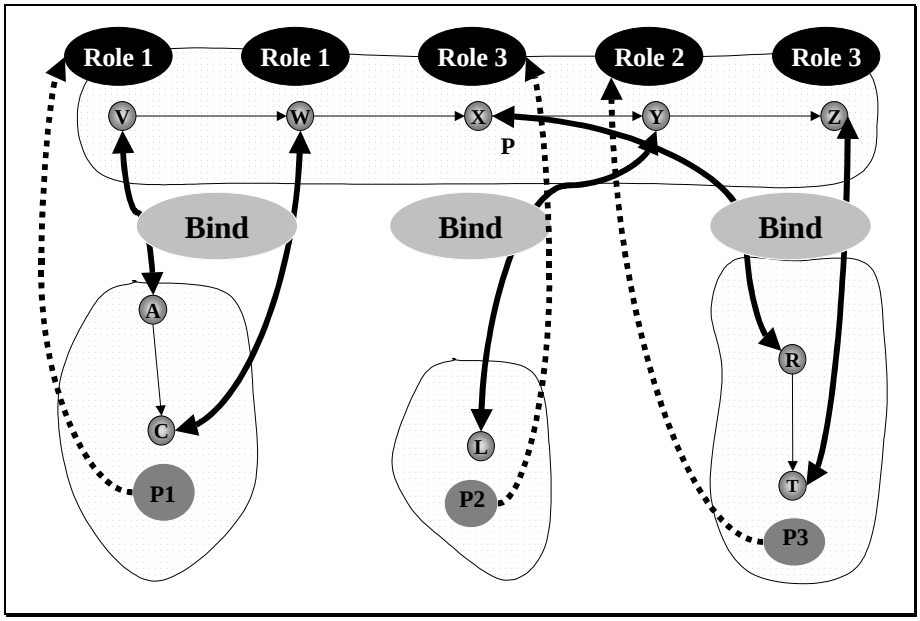
Note that flows produce no fault messages themselves. The lifecycle operations used to manage instances of a flow model return a fault message in erroneous situations (see Section 4.6.5 "Support for Lifecycle Operations"). As a future extension of WSFL we envision that a flow model will produce separate fault messages and that the data link construct will be allowed to materialize fault messages. This will allow in particular a conditional mapping taking into consideration whether or not an activity encompassed in a flow model returned a fault message or a regular output.

3.1.1.14 Instances and Models

A *flow model* is the specification of activities and their properties, as well as the associated wiring of the activities by means of control links and data links. An *instance* of a flow model or a *flow*, for short, results from navigating through the underlying graph of the instance or flow. And *navigation* means, to interpret a flow model according to the rules we sketched and that we will refine in Section 3.1.2 “Operational Semantics,” passing input to the instance, determining its start activities and performing them, receiving output of completed activities, determining their actual successors, materializing their input, performing them, and so on. Navigation actually results in assigning states to the activities, managing the context of the instance, and invoking operations.

3.1.1.15 Service Providers

A flow model specifies its requirements for services provided by a business partner when instances are made from the model. Often, the same business partner has to perform certain activities within a flow. And, often, the business partner has to offer the execution of these activities in a particular order. This expected behavior of a potential business partner is captured by the concept of a *service provider*. The service provider concept allows to specify the “role” that is expected to be played by a potential business partner.



The corresponding requirements are specified through a *service provider type*. A service provider type is just a named set of port types. The port types collected by a service provider type may be derived from the public interface of a flow model or may be just “opaque” port types. If a port type stems from the public interface of a flow model, the operations of the port type “inherit” restrictions from the flow model on the validity of invocation sequences between the operations from the port type. If a port type is opaque, no such restrictions exist. In this sense, flow models as well as port types represent a “type system” for service providers.

Thus, the support of port types from the public interface of another flow model will be specified (that is, the service provider is required to be of the corresponding service provider type) if the order of invocation of operations matters, and opaque port types will be specified if the order is immaterial. Note, that any mixture of opaque and “restricted” port types is allowed to be collected into a service provider type. Furthermore, by associating a set of activities of a flow model with the same service provider it is specified that at run time, operations of the same service have to be used for implementing the activities.

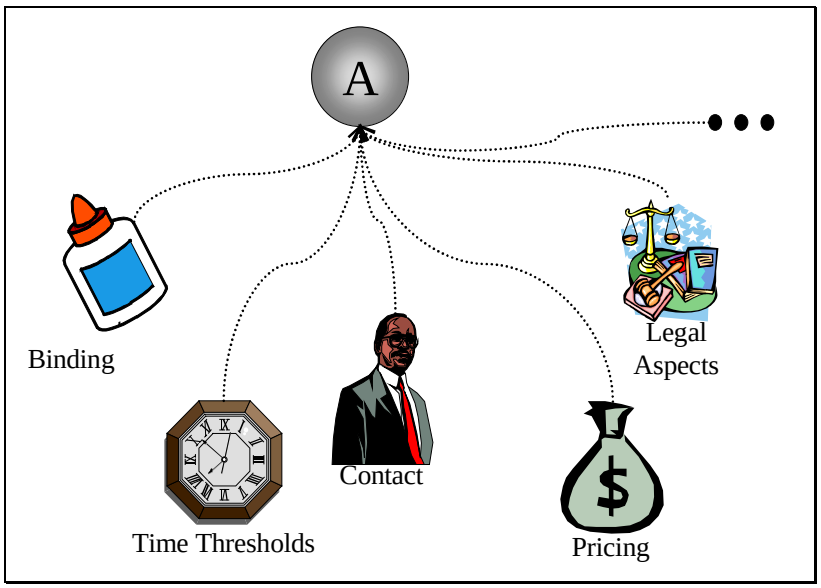
This might be perceived as specifying “roles” that potential partners must be able to play in order to do business with the organization that specifies an actual flow model. In order to actually select a specific service provider to bind to in the actual business context (that is, an “instance” of a role), so-called locators will be used: A *locator* is a specification of how to find a specific service provider. It can be a static locator that binds to a fixed service provider (“always bind to my preferred ticket agent”), it can be a “mobility” locator that allows to specify the actual service provider through messages exchanged as late as when the corresponding port is to be invoked (“send the bill for the ordered goods to the following e-mail address”), or it can be a query that further restricts in a declarative manner the set of possible partners playing the role needed (“bind to the provider that needs the shortest processing time”).

Actual service providers can be selected and bound to at different points in time: when a flow model is instantiated, when the first of a set of activities associated with a service provider is visited by the control flow, or whenever an activity is visited by the control flow.

In the figure above, partner P has a flow in which activities V and W need a service provider that can play role 1. In particular, V and W must be executed in the specified order. Partner P1 is such a service provider, offering implementations A for V and C for W performed in exactly the required order. The same is true for partner P3, who can play role 3 required in P’s flow. Partner P2 offers simply a port type without any ordering constraints: L implements activity Y. The dynamics of the binding are not shown in the figure.

3.1.1.16 Endpoint Properties

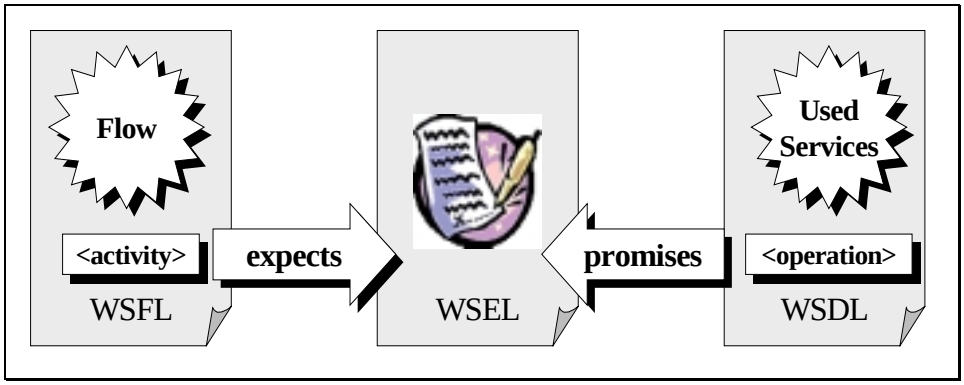
Typically, activities represent business tasks and interactions between trading partners (that is, service providers). As such, they do have additional business semantics described by properties like legal obligations of each partner side, costs and prices for performing an activity, maximum duration and maximum number of retries, actions (for example, escalations) that should happen if such thresholds are exceeded, contact points who are in charge at both trading partners, security aspects (confidentiality, non-repudiation), at-least-/at-most-/exactly-once execution, and so on. These properties are simply referred to a s*endpoint properties*.



Service providers are determined dynamically through service locators. The “most appropriate” service provider has to be found, where “most appropriate” typically means different things in different situations. For example, “most appropriate” can designate the cheapest, or fastest, or most secure, or most reliable, or … provider of a service that implements an activity of a flow. To allow for the corresponding matchmaking, both activities

on one side and operations, port types, ports, or services on the other side must be described by endpoint properties.

As a consequence, endpoint properties are neither WSFL-specific nor WSDL-specific, and they have to be used within WSFL as well as WSDL. Because of this, we envision a separate language for describing endpoint properties, here called Web Services Endpoint Language (WSEL). WSFL foresees the usage of appropriate extensibility elements to describe endpoint properties of activities, and assumes that Web Services will be described through endpoint properties in such a way that matchmaking can be done through service locators (see the following figure).

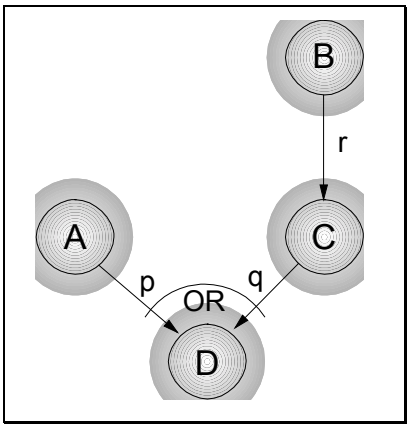


3.1.2 Operational Semantics

A significant part of the operational semantics of our metamodel has already been discussed when the various ingredients of the metamodel have been introduced. The most significant piece missing is *dead-path elimination*, which is discussed in the next section, 3.1.2.1 “Death-Path Elimination.” Finally, in Section 3.1.2.2 “Summary: Operational Semantics,” we summarize the operational semantics in the format of a list.

3.1.2.1 Dead-Path Elimination

The following figure depicts a flow with start activities A and B. Thus, when the flow model is instantiated, activities A and B will be scheduled to be performed. Assume that A completes and that the transition condition p evaluates to true. Navigation will wait until the truth-value of the transition condition q is available before deciding whether or not the join activity D has to be performed (assuming that full synchronization at the join node is required).



Now, assume that activity B completes and that the transition condition r evaluates to false. In this case, activity C will never be performed. But if C will not be performed, transition condition q will never be evaluated. And, thus, the join condition of D will never be evaluated

and D will never be performed. This erroneous situation is avoided through *dead-path elimination*.

Dead-path elimination has to take place whenever it becomes clear that a particular activity will never be performed. This is the case when a join condition of a join activity evaluates to false, or when the transition condition of an activity with exactly one incoming control connector evaluates to false. Originating from such an activity, dead-path elimination has to traverse the underlying flow model's graph until the next join activity or end activity is reached.

During this traversal, all visited transition conditions have to be assigned a truth-value of false and have to be marked as evaluated. Assume that a join activity is reached: When the associated join condition can already be evaluated, this is done; if the join condition evaluates to true, the join activity can be performed, otherwise dead-path elimination continues. When the join condition of the join activity reached cannot be evaluated yet, the decision about whether or not the join activity will have to be performed is deferred.

3.1.2.2 Summary: Operational Semantics

At this point, we discussed all the fundamental ingredients of our metamodel and their operational semantics. The following list summarizes this operational semantics by giving you a sketch of the navigation algorithm used to interpret flow models:

1. When a new instance of the flow model is created, determine all start activities of the graph, map the flow source to the input of the start activities, and perform them.

   From then on, the navigation takes place whenever an activity implementation returns, that is:

2. When the activity implementation returns, compute the exit condition of the activity.

   a. Determine the actual values of all formal parameters of the Boolean expression corresponding to the exit condition.

   b. If the exit condition evaluates to false, repeat the execution of the activity implementation with the same materialized input at the actual service provider chosen.

3. If the actual activity completed, all control links leaving the activity are determined.

4. The transition conditions of these control links are determined.

5. The actual values of the formal parameters of the Boolean expressions corresponding to the transition conditions are computed.

6. The truth-values of these Boolean expressions are computed based on the actual parameter values.

7. The endpoints of these control links are determined (that is, the *potential* follow-on activities of the completed activity are computed—we have to visit even endpoints of control connectors with false transition conditions to determine whether these are *actual* follow-on activities in case these are join activities).

8. Determine all non-join successors within the set of potential successors whose incoming transition condition has been evaluated to false, and determine all join successors within the set of computed successors whose join condition evaluate to false (that is, the subset of "dead activities" within the computed set of successors is determined).

a. Perform dead-path elimination originating at each of these nodes.

b. Each join activity reach that has a join condition that evaluates to true enters the state “enabled.”

9. Determine all non-join successors within the set of potential successors whose incoming transition condition has been evaluated to true.

   All activities from this computed set enter the state “enabled.”

10. Determine all join activities within the set of potential successors whose join condition evaluate to true.

    All activities from this computed set enter the state “enabled.”

11. All other successors remain in their current state and must wait for a future navigation step for a possible state change.

12. Determine the actual service provider for all activities that entered the state “enabled” based on the steps before, that is, for each of these activities:

    a. Determine its associated service locator.

    b. Evaluate the service locator.

    c. Chose an actual service provider.

13. Compute the input message of each “enabled” activity.

    a. Determine all data links that target such an activity.

    b. Determine the output messages or message parts referred to in a map of the data link.

    c. Apply the maps to materialize the input message.

14. Perform the operation with the materialized input at the actual service provider chosen if the operation is an *in* or *in/out* operation. For *out* or *out/in* operations similar processing has to be done.

15. If all other end activity has already completed, the flow has finished. Otherwise, resume navigation.

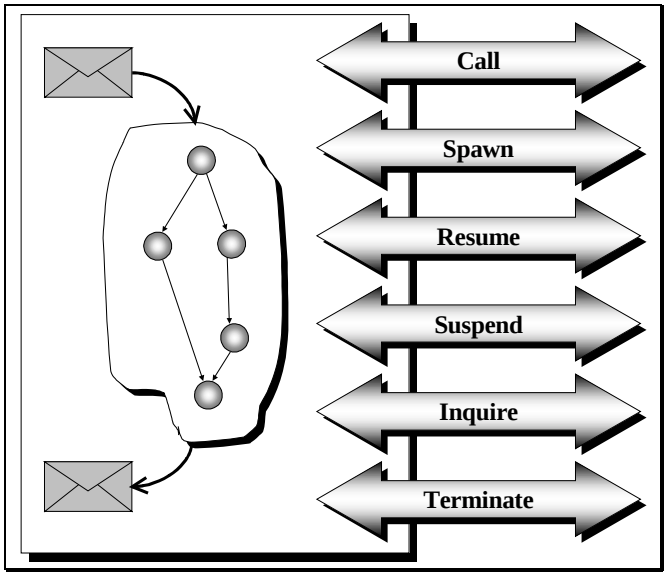
16. If the flow has finished, compute the flow sink based on the data links pointing to it.

3.2 Lifecycle Interface

Each flow model is always associated with a port type that allows managing the lifecycle of instances of the flow model. For example, the port type provides operations for instantiating the flow model and immediately executing it (`spawn` operation), for suspending and resuming a given instance (`suspend` and `resume`), for querying flow model information (`enquire`), and so on. It is assumed that each service provider supporting a particular flow model does provide ports that implement this port type by means of WSDL.

An instance of the flow model is represented by a unique instance ID. The instance ID is used by lifecycle operations that operate on an instance of a flow model, for example, `suspend`. The instance ID is returned by the `spawn` lifecycle operation. The input message

(source) and the output message (sink) of a flow model are part of the signature of the lifecycle operations (where applicable–see Section 4.6.5 “Support for Lifecycle Operations”).



The lifecycle operations can be used to include a flow model as an activity implementation in another flow model. The `spawn` lifecycle operation allows starting a new flow model as an activity step in another flow model. As a result, a new, independent flow is started that is running independently. The `call` lifecycle operation allows running a flow model as an activity in another flow. The end of the `call` operation marks the end of the activity and the flow model navigation continues.

A flow model can have input and output data associated with it, for example, a supplier flow model receives as input the ordering information and returns as output the information about successful completion. In order to map the flow model input data to activities, a `wsfl:flowSource` element is used to represent the incoming data in the flow model. A `wsfl:flowSink` element is used to represent the flow model output data and can be used to map the output data of activities to the output data of the flow model (see Section 4.5.3 “Data Links and Data Mapping” for more details on `flowSource` and `flowSink`).

The name of the lifecycle port type and operations are defined in the flow model definition. The lifecycle operations that are supported by a flow model are defined by an export element for every supported lifecyle operation. The following lifecycle operations are defined in WSFL and can be exported:

- `spawn` - Creates an instance of the flow model and starts it. The operation returns as soon as the flow is started, passing the instance ID of the new flow instance as a result.

- `call` - Creates an instance of the flow model and executes it. The operation returns only after the flow instance has completed, passing the output message of the flow as a result.

- `suspend` - Suspends the flow model instance, that is, temporarily interrupts the execution of the flow instance.

- `resume` - Resumes the flow model instance, that is, continues the execution of a flow instance that was previously suspended.

- `enquire` - Queries the status of the flow model instance. As a result, the status of the flow model instance is returned.

- `terminate` - Terminates the flow model instance.

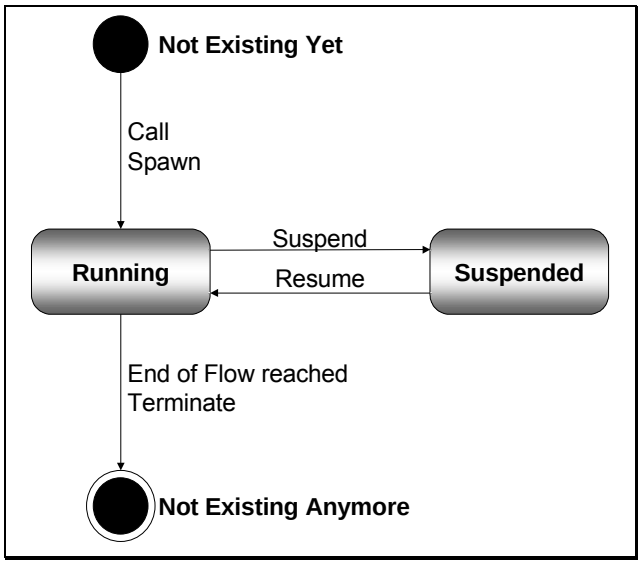
As a future extension, we envision an “observer” concept that allows specifying to whom the outcome of an operation is to be sent. For example, the instance created and started by the `spawn` operation might finish long after the entity that issued the operation has ceased to exist. Registering another entity to which the result of the instance has to be sent is a valuable potential future extension.

3.3 Business Process Lifecycle

A flow instance is created through one of the lifecycle operations (`spawn`, `call`). After its creation, the new instance will begin to be performed, which is reflected by the associated state of “Running.”

Once the complete flow has been executed (that is, all the activities have been visited as prescribed by the control links), the flow instance will cease to exist. When in state “Running,” the same can be achieved through the lifecycle operation `terminate`.

Finally, running flows can be suspended through a respective lifecycle operation (`suspend`), and resumed again (`resume`). When suspended, individual activities may be completed, but navigation through the flow instance is no longer advanced by the container managing it (the “flow engine”). After resuming a suspended flow instance navigation continues where it was left, considering activities that completed in between.



3.4 Activity Lifecycle

When a flow is instantiated, it might be perceived that all of the activities included in the associated flow model are instantiated, too. Such an activity instance represents the usage of the activity in the concrete instance, for example, the actual service provider supporting it, and so on.

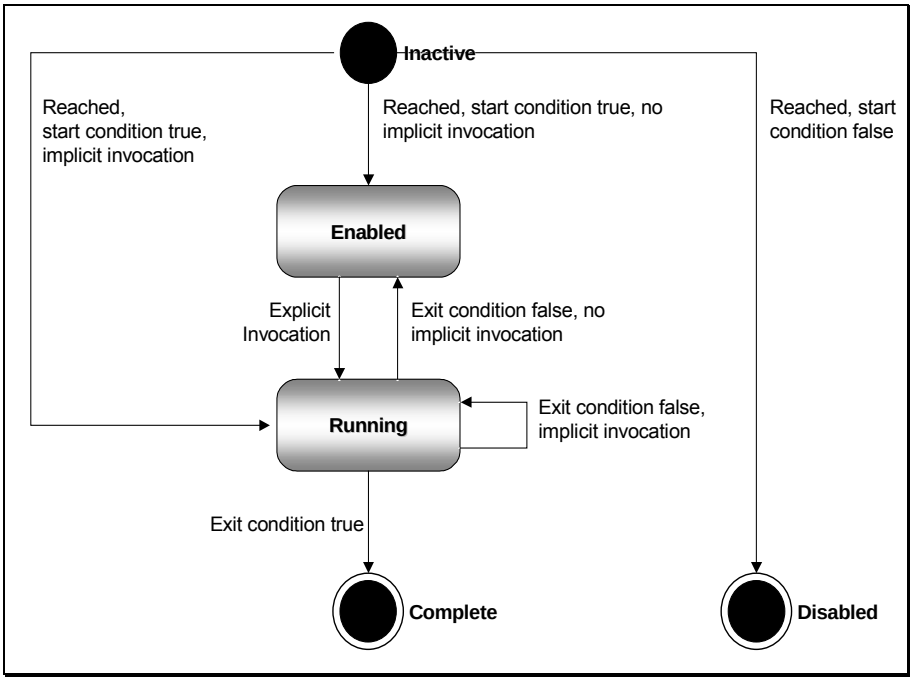
Activity instances do have states assigned. When an activity is instantiated, it assumes its initial state of “inactive” until it is reached by the control flow. Once an activity is reached by

the control flow, there are two basic possibilities:

- The flow engine decides, based on the activity's join condition, that this activity will never be executed, and therefore puts it into state "Disabled." Dead-path elimination will take place for all paths leaving that activity (see Section 3.1.2.1 "Dead-Path Elimination").

- The flow engine decides that this activity instance could now possibly be executed and puts it into state "enabled." Depending on the nature of the activity and its associated operation, it might remain in that state until it is started through an explicit request (for example, for *in* or *in-out* operations), or the flow engine will start it right away (for example, for *out* or *out-in* requests). Once started, its state will be "Running."

After the associated operation has completed, continuation of the activity depends on its exit condition. If this evaluates to false, the activity is iterated, by either continuing with "enabled" or "running," depending on the associated operation. If the exit condition evaluates to true, the activity reaches the state "Complete."



3.5 Recursive Composition Metamodel

We will now describe how new Web Services are composed out of existing ones. The base for the composition metamodel of WSFL is port types and their grouping into service provider types. Operations of service provider types' port types are linked together to specify the potential interaction between the corresponding business partners. Because each flow model defines a service provider type, the composition metamodel includes in particular the means to define the interactions between flows. We first give an overview of the composition metamodel and provide more details afterwards.

3.5.1 Composition Metamodel Overview

In previous sections, we have seen how a business process is represented as a flow in the metamodel of WSFL. Creating a flow model involves defining activities and creating a model of execution by linking the activities using control and data links. The flow metamodel

described so far is the central abstraction used to represent the usage of existing programs and Web Services in a business process and, in turn, to represent a business process as a Web Service.

3.5.1.1 Global Models

In addition to the flow metamodel, WSFL relies on a simple recursive composition metamodel to represent the ability to describe the interactions between existing Web Services and to define new Web Services as a composition of existing ones. We use the term *global model* for a model defined using the composition metamodel. From the perspective of the composition metamodel, a business process described by a flow model is just another Web Service.

A *service provider type* defines the interface of a Web Service; it represents a set of port types that declare the operations supported by the Web Service. Service provider types provide the building blocks for flow models as well as for global models; both models provide complementary views on the overall interactions between service provider types.

A flow model defines the flow of control and data between a set of activities. Each activity is associated with an instance of a service provider type, a *service provider.* This service provider is responsible for the realization of the activity. Essentially, an activity defines the requirements of the flow model on some service provider; the actual service providers are chosen based on *locators* (see Section 3.1.1.15 “Service Providers”). A flow model itself defines a service provider type, and the requirements of the encompassed activities on external Web Services define a significant portion of the external interface of that service provider type.

A global model defines the interaction between a set of service providers. Interactions are modeled using *plug links* between “dual” operations on the service provider types involved in the composition. For example, a notification operation on one service provider can be “plug linked” to a one-way operation on another service provider, or a solicit-response operation can be “plug linked” to a request-response operation.

The relation between the global and flow metamodels can then be stated as follows: The flow metamodel supports to describe the internal behavior of Web Services, while the global metamodel supports to define the interactions between Web Services. The two aspects are very closely related, though, and the following sections describe their relationship.

3.5.1.2 Service Providers as Components

We have seen that service providers represent the functionality that a flow model requires from other business partners. Service providers have a type (a service provider type) that defines their external interface; again, the service provider type may actually represent the external interface of a flow mode,l and this information can be used when the sequencing of the interactions with a service provider is important.

In either case, a service provider presents a *public interface* in the form of a set of port types, which define the ways in which a service provider can interact with other providers. Essentially, service providers are “peer-to-peer” partners from the perspective of the global model, and the global model facilitates the “match-making” between these peers. For example, a solicit-response operation defined by another service provider defines the requirements on a matching request-response operation to be provided by another service provider.

Service providers are the units or building blocks of any composition from the composition metamodel perspective. A composition consists of a set of connected service providers, which may in turn become a Web Service and be used as a new service provider in other compositions.

3.5.1.3 Connections between Service Providers

The interaction between service providers is conducted through the operations defined by their public interfaces. These operations need to be connected to each other for the interaction to take place; source and target operations of the connection must have “dual” signatures: a solicit-response operation can be connected to a request-response operation, and notification operations to one-way operations. Interactions between service providers are represented in the composition metamodel of WSFL as a special type of link, a *plug link*.

In a way, a plug link can be interpreted as an invocation of a “serving” operation by a “requesting” operation; this analogy is especially useful for request-response/solicit-response operation pairs. A plug link can also be interpreted as event propagation (in the messaging sense) between two components: the source component sends a triggering event to the target and thereby triggers some action to take place on the receiving end. The receiver may or may not respond to this event. The latter interpretation is useful for notification/one-way interactions.

3.5.1.4 Flow Models as Service Providers

The previous section explained the general concept of defining interactions between any kinds of service provider. It is important to note that from the perspective of the global model, it is irrelevant whether or not a WSFL flow model defines the “internal behavior” of a service provider. It is also important to understand how the service provider type of a flow model is related to the internals of a flow model.

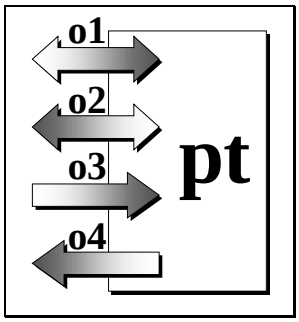
A flow model constitutes the definition of a new service provider type. The new service provider type has at least one port type in its public interface, including the lifecycle operations described in Section 3.2 “Lifecycle Interface.” It can have additional port types that represent the requirements of the flow model on other business services that are used to realize certain activities in the flow model.

For each activity in the flow model that requires an external service provider for its realization, the external interface of the flow model defines one operation. The association between the activity and this operation is defined using the *export* element in the activity definition. The operation representing the activity defines the activity’s requirements on some other Web Service to qualify as a realization of the activity. The actual association between the activity and the realization is done in a global model that defines the interactions between both Web Services through a plug link.

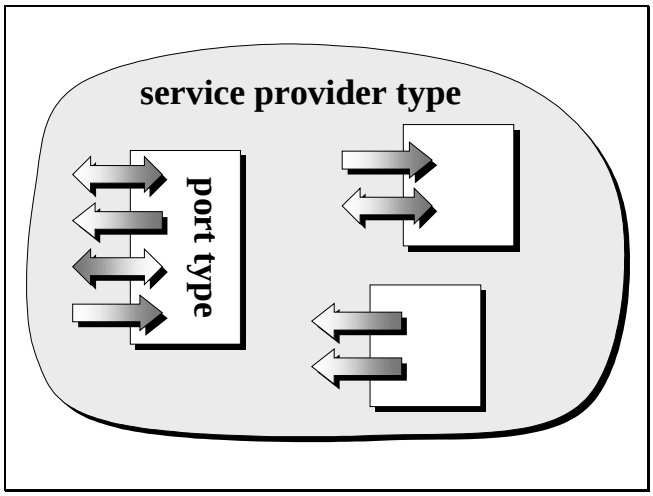
3.5.2 Graphical Representation of Port Types and Service Provider Types

In the following figure, we are representing port types as a rectangle and a set of arrows with one or two heads. Note that the graphical representation suggested here is used for illustration purposes only; WSFL modeling tools may use any representation they like. Each operation is represented by a separate arrow. The following figure shows a port type called pt with four operations. By convention, a two-headed arrow represents the head for the direction of the first “stimulating” interaction as dark-shaded, while the head for the second interaction is not shaded.

Thus, operation o1 is a request-response operation, that is, a supporting endpoint will receive a message and it will respond with a correlated message. Operation o2 is a solicit-response operation, that is, a supporting endpoint will send a message and it expects a correlated message afterwards. Operation o3 is a one-way operation, that is, a supporting endpoint receives a message. Operation o4 is a notification operation, that is, a supporting endpoint sends a message. Note that shading of the arrow head is actually not needed for one-way and notification operations. Furthermore, shading is omitted completely in case the kind of operation is irrelevant.



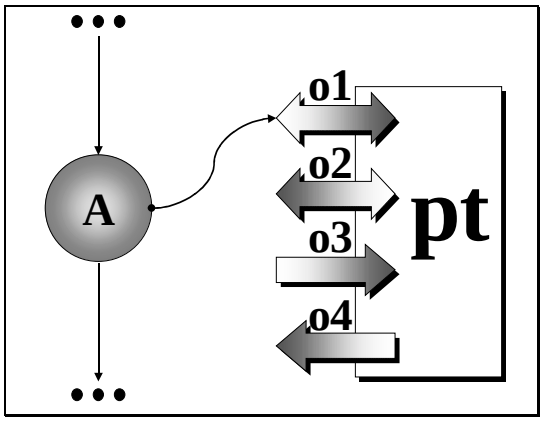
Service provider types are named collections of port types. Any mixture of “opaque” port types (that is, port types whose operations are not known to be constraint by an associated flow model) and port types that stem from flow models is allowed. Concrete service providers are determined through locators that can be associated with service provider types. The next figure shows the graphical representation of a service provider type; we will use the same representation for concrete service providers, too.



3.5.3 Operations As Activity Implementations

One goal of WSFL is to enable Web Services as implementations for activities of business processes. For this purpose, an activity may refer to an operation of the service provider type’s port type that defines the external interface of the flow model to specify which kind of service is needed at run time to actually perform the business task represented by the activity. The next figure shows a flow in which an activity called A is implemented by a service that realizes operation o1 of port type pt (we will write pt.o1 to indicate that o1 is an operation of port type pt).

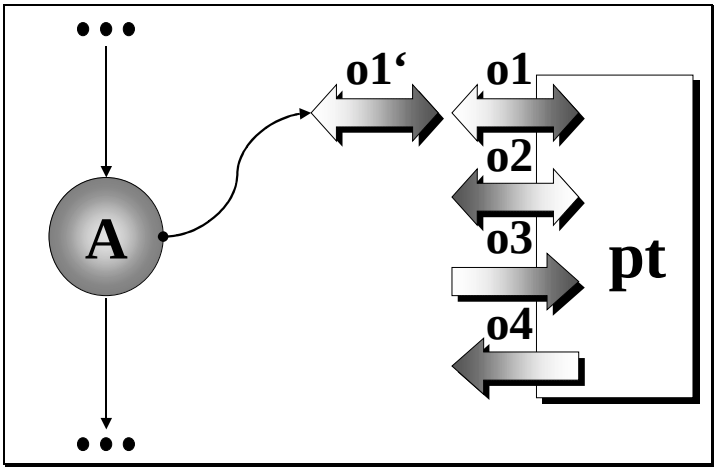
At run time, when navigation proceeds to A, a concrete port is chosen that provides an implementation of port type pt and operation o1 and a corresponding binding is used to actually invoke this implementation. The actual port chosen is basically determined through the service provider and locator construct of the metamodel.



3.5.4 Which Operation Is the Activity Implementation?

In a first attempt, one could directly specify port type pt and operation o1 from the service provider type to interact with the service needed as implementation of activity A. From a modeler's point of view, this would be fine: We envision that a flow modeler will interact during build time with UDDI (or a similar directory) searching for businesses that are supporting implementations of services implementing various activities of the flow actually being modeled. Simply, the modeler wants to "drag" the operation of a port type found to the activity whose appropriate implementation is looked for, and "drop" it there. This corresponds in a natural manner to specifying the operation and the port type found directly as the activity's implementation.

But doing so would ignore the fact that the other goal of WSFL is to support a composition model for Web Services: A flow model should be viewable as a new *stand-alone*, a *self-contained* Web Service describing all of its interaction requirements and offers. Thus, an activity of a flow model should not link directly to the proper port type and operation that provides its implementation. Instead, a flow model should describe all of its interaction requirements with port types and operations in such a manner that the entity representing the flow model (or the new Web Service, respectively) builds a new service provider type.



To achieve that, the flow model specifies as implementation of an activity an operation that is dual to the one providing the implementation proper. For example, if the activity implementation proper is a request-response operation, the dual operation specified as activity implementation is a corresponding solicit-response operation. Again, not the implementation proper but a dual operation is specified as the activity's implementation. This dual operation may be perceived as a "proxy" for the implementation proper.

In the figure above, operation o1' is the solicit-response operation dual to the request-response operation o1 of port type pt. While pt.o1 represents the proper implementation of activity A chosen by the flow modeler, it is operation o1' that is specified as A's implementation, that is, o1' is the proxy for pt.o1.

3.5.5 Realizing Activity Implementations

Note that a realization compliant with the WSFL metamodel does not necessarily have to enforce to really provide an implementation of dual operations, that is, proxies (as ports or somehow else). In fact, if the proper implementation of an activity is a request-response operation or a one-way operation, the corresponding dual solicit-response or notification operation, respectively, may not be implemented explicitly by a programmer but implicitly by the flow engine itself. The flow engine itself might send a message to the chosen port providing a binding for the request-response operation and expect the corresponding result back, if applicable.

For example, the activity `getStockQuote` is implemented by the `sendQuote` operation of the `stockFacilities` port type. The `sendQuote` operation is a request-response operation that expects a trade symbol as input and that produces the actual stock price as output. Service provider `MyExchange` provides the port `MyExchangeServices` that in turn implements a Simple Object Access Protocol (SOAP) HTTP binding of the `stockFacilities` port type. The flow engine might be capable of communicating directly with the port sending a corresponding SOAP request to the port and receiving the SOAP response. There is no need for a programmer to implement the solicit-response operation that performs the communication with the `MyExchangeServices` port.

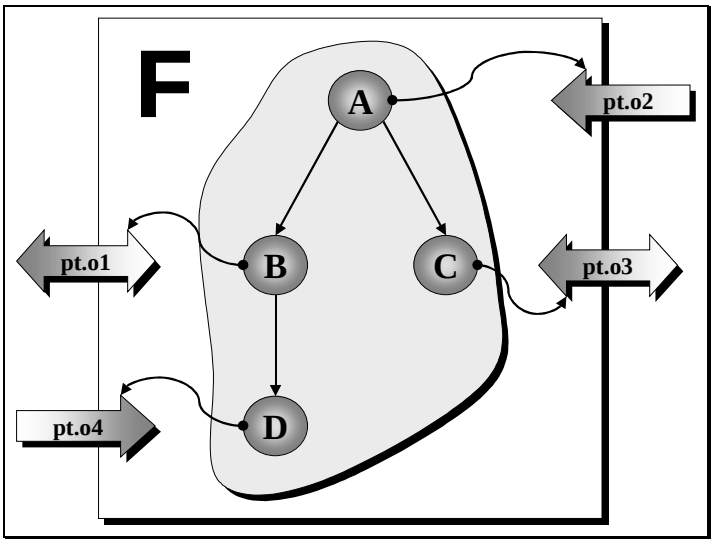
Thus, it is absolutely valid that realizations of solicit-response or notification operations implementing activities within flows are deferred to a generic implementation of the realizing flow engine or underlying middleware. And it is this generic component that performs the required interaction with the proper implementations of request-response and one-way operations.

This supports our vision for easily deploying Web Services when modeling flows. When the modeler chooses a request-response or one-way operation found in for example Universal Description, Discovery and Integration (UDDI) as the implementation of an activity, he or she may request from the modeling tool to generate a dual operation and defer its implementation to a generic realization (for example, the underlying flow engine). This dual implementation will then appear as the ("virtual" or "proxy") implementation of the activity in WSFL.

3.5.6 Exporting Operations

In order to allow flow models to be defined as service provider types (that is, a set of port types), the metamodel provides a construct to *export* operations implementing encompassed activities. These exported operations are grouped into port types that define the public interface of the flow model. All those operations will be exported that require interactions with some external service provider. These operations specify the interaction offers and demands with respect to the outside world. Stated another way, activity implementations that do not require communication with the outside do not need to be exported.

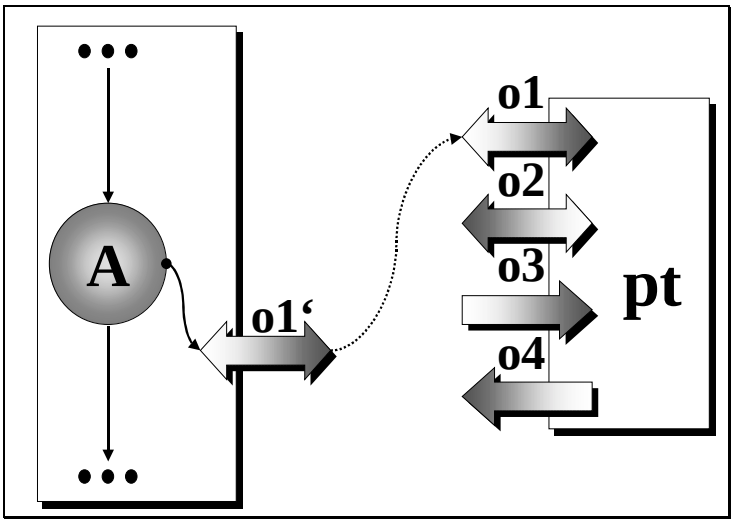
In the figure below, flow model F encompasses the set of activities (A,B,C,D). Each of these activities is implemented by an operation of some port type (these port types are specified with the corresponding activities and are not shown in the figure). These operations are exported and aggregated into a new port type called pt in the figure. This port type pt represents the external interface of the flow model F, that is, the modeler has chose to define a single port type to make up the service provider type representing the flow model.



3.5.7 Plug Links

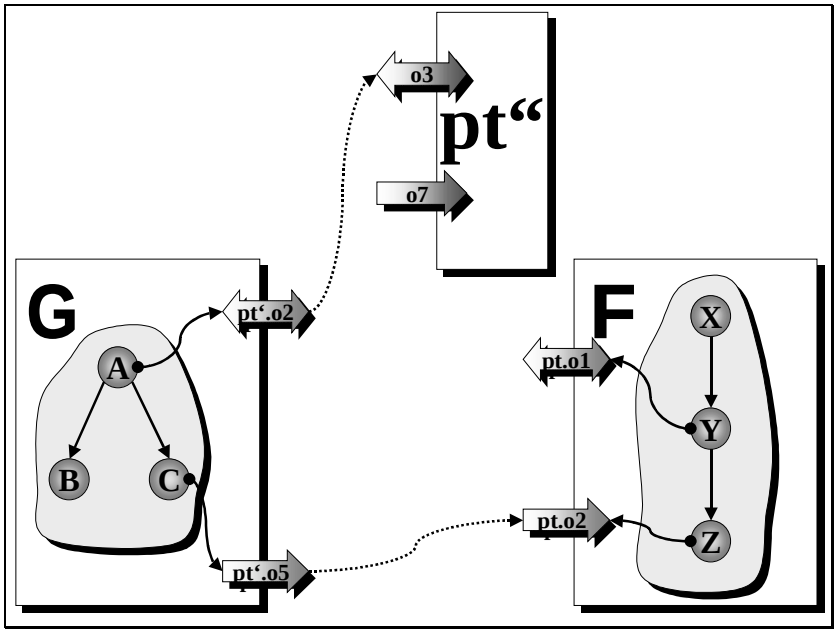
What is still missing until now is the specification of which exported operation of a flow model interacts with which (dual) operation of a port type of an external service provider. Thus, we have to provide a means to specify pairs of dual operations to indicate the interaction between service providers actually going on at run time. This construct is called a *plug link* in the metamodel. A plug link connects two dual operations of different port types indicating that a corresponding interaction has to take place in order to completely implement an activity. Note that both plug-linked port types are allowed to be opaque, that means, not related to a flow model at all; then, of course, the plug link does not make any assumptions about implementing an activity but just specifies the interaction requirement between both port types.

In the following figure, the proper implementation of activity A is the request-response operation o1 of port type pt. But in order to describe the service provider type representing the flow model that includes activity A, the operation o1' dual to pt.o1 (that is, o1' is a solicit-response operation) is specified as A's implementation at the language level and is exported to an appropriate port type of this service provider type. A plug link is specified between o1' and o1 to indicate the interaction between o1' and o1. The plug link means that o1' will initiate the interaction by sending a message to the port implementing pt.o1 through the binding chosen. Operation o1 will receive the message as a request and perform whatever action has to take place to produce the result message. The latter message is sent as a response message to the implementation of o1', which in turn consumes this message. If applicable, this message will typically be returned to the flow engine as the result of the execution of the implementation of activity A.



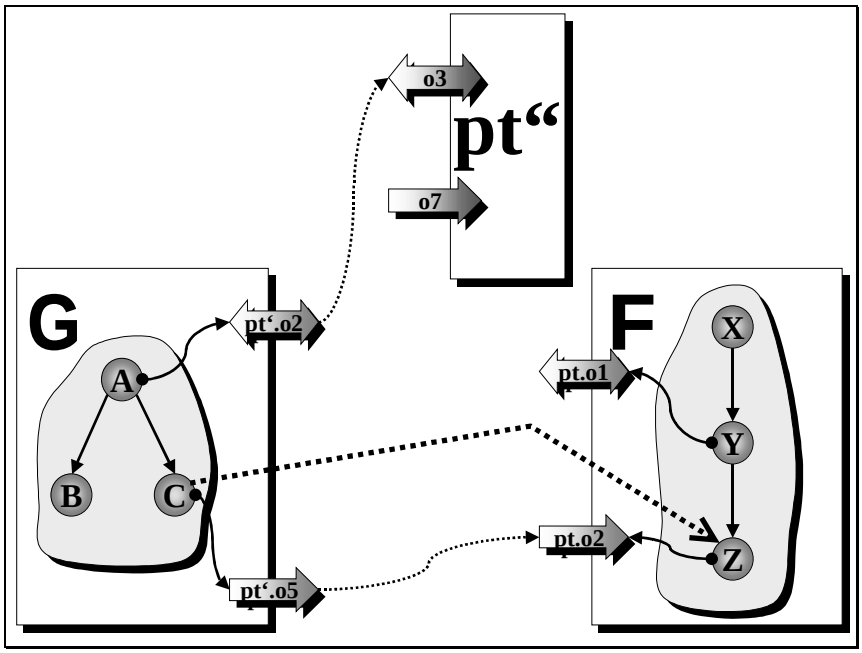
3.5.8 Flows and Plug Links

As discussed, the WSFL metamodel provides a mechanism to define flow models as new service provider types. Because operations of any kind of port types can be used as implementations of activities of a flow, especially each operation exported by a flow model can be used as such an implementation. In the following figure, flow models F and G are assumed to define service provider types that consist of a single port type each for simplicity. Activity C of flow model G is implemented by pt'.o5, where pt' is the port type describing the public interface of G. But the proper implementation of C is operation o2 of port type pt. Port type pt, in turn, is the public interface of flow model F. As a net effect, a plug link is established between operations (namely o5 and o2) of port types associated with two flow models (namely G and F). The same figure also shows that activity A of flow model G is implemented by pt'.o2, which in turn is plug-linked to operation o3 of the (opaque, that is, non-flow) port type pt".

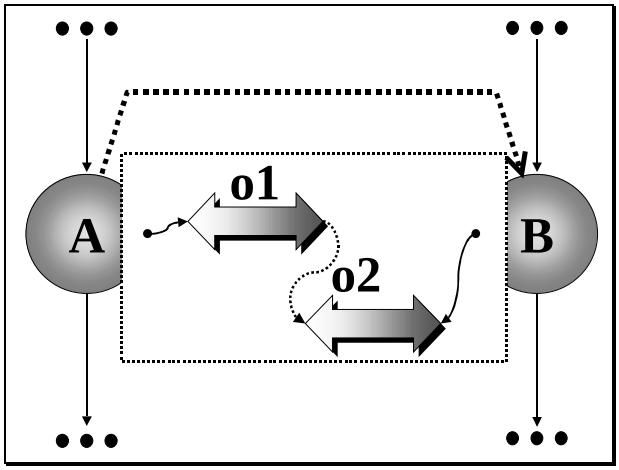


3.5.9 Making Things Convenient

The metamodel provides a facility to ease the specification of interactions between operations of different flow models: Conceptually, a plug link can be specified that simply connects two activities of two different flow models. In the following figure, activity C of flow model G is connected through a plug link to activity Z of flow model F. Such a plug link between activities is just a syntactical convenience: If a tool allows to draw such a link, it is always assumed that the tool will automatically generate the associated export constructs and the associated plug links for the exported operations. In other words, at the language level, no plug link between activities exists.



Assume a tool allows drawing a plug link between activities like A and B in the following figure. Then, the tool will generate the export of the request-response operation o2 (the proper implementation), the derivation of the proxy operation o1 (the solicit-response operation) and its export, as well as the plug link between o1 and o2. This generation action of the tool might be referred to as “activity plug link explosion.” In this case, “explosion” means substitution of the fat dashed arrow between A and B by the constructs within the dashed box.



3.5.10 Mapping Data

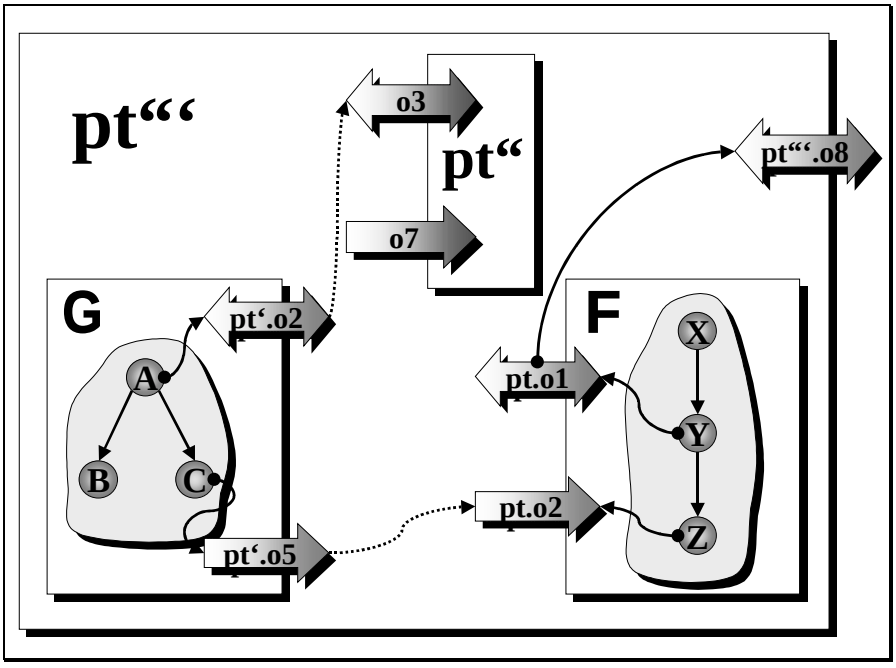
The WSFL metamodel does not assume that the signature of an activity and its implementing operation as well as the signature of two plug-linked operations are identical. Because of this, both the export construct as well as the plug link construct allow to specify a data mapping between the source and the target of an export and a plug link, respectively. This is very similar to the data-mapping capability of data links discussed before.

3.5.11 Aggregating Web Services

The composition metamodel of WSFL allows exporting operations from a port type of one service provider to a port type of another service provider. Effectively, this represents the delegation of the implementation of the target operation of the export to the source operation of the export. In particular, this allows a group of collaborating service providers to appear as a single new service provider to the outside world.

For example, when modeling flows and assigning implementations to activities it might occur that some exported operations will not be coupled with operations of port types already chosen in the composition process. These “dangling” operations do need other port types and operations in order to result into a complete realization of the final flow model or service provider type, respectively. The export feature above allows catching such dangling operations and associating them with new port types.

In the following figure, the operation o1 implementing activity Y of flow model F is dangling: It has no matching operation of a service provider assigned, that is, the overall implementation of Y is “incomplete” (we do assume that o1 is not fully implemented internally by the provider of flow model F). Thus, a new port type pt”’ is defined that has an operation o8, and o1 is exported to operation o8 of the new port type pt”’. The operations of pt”’ can be used as endpoints of plug links, thus completing the overall end-to-end implementation at a later time. Note that one might perceive that service providers of G, F, and pt” cooperate to provide a new service represented by pt”’ and offer it to their joint customers.

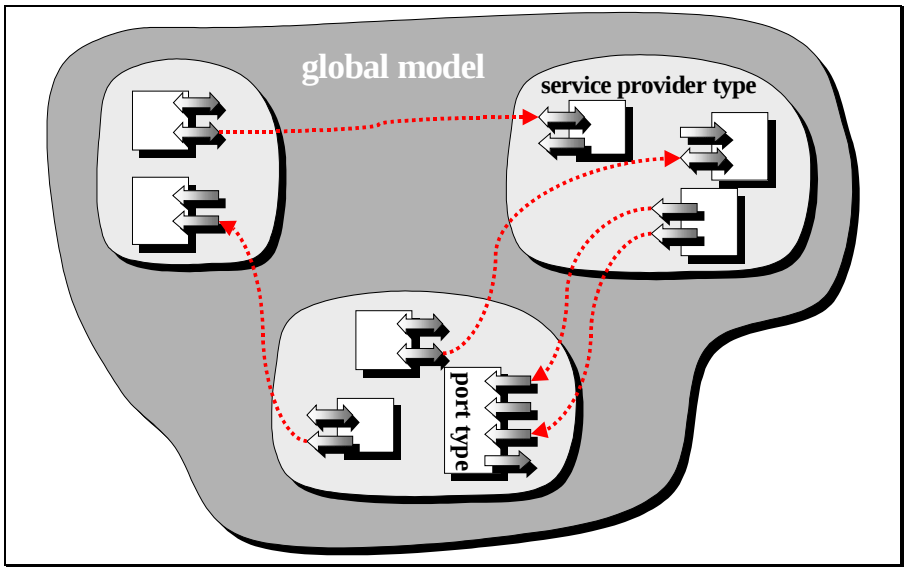


3.5.12 The Global Model

The interaction patterns between service providers or service provider types are specified in the WSFL metamodel as global models. A *global model* consists of a set of service providers or service provider types, respectively, as well as corresponding plug links and exports between operations of the encompassed port types.

A global model may specify a new service provider type. This service provider type can be used in turn in a composition through a flow model or another global model, respectively. Thus, the composition model of WSFL is recursive in nature.

Note that the service providers or service provider types aggregated within a global model are not necessarily associated with a flow model. In particular, it is even perfectly valid that all of the service providers or service provider types within a global model are opaque, that is, not associated with a flow model at all. As such, a global model specifies all the possible interactions between the affected service providers during run time.



4 Language Description

In this section, we are describing the Web Services flow language in its details. In Section 5 “Appendix A: WSFL Schema,” we present the XML schema of the language.

4.1 Document Structure and Naming

A WSFL document contains the definition of one or more flow models and global models. In addition, WSDL `portTypes` can be optionally defined inside a WSFL document if required. Finally, a WSFL document can have import clauses referring to other WSDL and WSFL documents. These elements are described in detail in the following sections. At the root of the WSFL document there is always a `<definitions>` element.

A WSFL document can be identified by the optional `targetNamespace` attribute encoded on the root `<definitions>` element. If included, the value of the `targetNamespace` attribute is an absolute Universal Resource Identifier (URI). This URI is used to identify elements defined in the WSFL document when referenced from other WSFL document (see Section 4.2 “References to External Definitions”).

The structure of the document is represented in the schema language [2] by the following complex type definition:

```
<complexType name="definitionsType">
   <sequence>
      <element name="import" type="wsfl:importType"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name=”serviceProviderType”
               type=”wsfl:serviceProviderTypeType"
               minOccurs=”0” maxOccurs="unbounded"/>
      <element ref="wsfl:flowModel"
               minOccurs=”0” maxOccurs="unbounded"/>
      <element ref="wsfl:globalModel"
               minOccurs=”0” maxOccurs="unbounded"/>
   </sequence>
   <attribute name="targetNamespace" type="uriReference"/>
</complexType>
```

The complete schema for WSFL can be found in Section 5 “Appendix A: WSFL Schema.”

4.2 References to External Definitions

WSFL makes the assumption that the services used in a composition have been described using WSDL (for interface and deployment definition) and WSFL (for service provider types, flow models, and global models). For this reason, a WSFL document usually includes references to WSDL and WSFL definitions contained in other documents.

References to external definitions are made using *qualified names*, which are represented by the QName schema type; see [3]. A qualified name contains a namespace part and a local part. The namespace part matches the `targetNamespace` attribute of the document where the referenced element is defined, while the local part corresponds to the value of the `name` attribute of the element.

As in WSDL, it is possible to associate a namespace with a document location by using the `<import>` element. The usage and syntax of this element is completely similar to the known WSDL element, see [1].

4.3 Flow Models

A service composition is represented in WSFL by a `<flowModel>` element, which is named using the `name` attribute. The definition of a flow model includes two kinds of information: the specification of how the composition uses the services being composed to create a flow model, and the definition of the service interface provided by the composition.

The public service interface of the flow model is specified as service provider type in the `serviceProviderType` attribute. Implementations of activities from the flow model defined can be exported by a corresponding `<export>` element to an operation of one of the port types of this service provider type. Details on the definition of the service interface of a flow model are provided in Section 4.6 ”Defining the Interface of a Flow Model.”

The flow model proper is defined using six different elements:

- The `<flowSource>` and `<flowSink>` elements define the input and output of the flow model.
- The `<serviceProvider>` elements represent the services participating in the composition.
- The `<activity>` elements represent the usage of individual operations of a service provider inside the flow model.
- The `<controlLink>` and `<dataLink>` elements represent control and data connections between activities in the model.

Service providers are discussed in Section 4.4 “Service Providers and Service Bindings,” activities and links in Section 4.5 “Defining Business Processes.”

The schema syntax for the `<flowModel>` element is provided in the following code sample:

```
<complexType name="flowModelType">
   <sequence>
      <element name="flowSource"
               type="wsfl:flowSourceType"
               minOccurs="0"/>
      <element name="flowSink"
               type="wsfl:flowSinkType"
               minOccurs="0"/>
      <element name="serviceProvider"
               type="wsfl:serviceProviderType"
               minOccurs=”0” maxOccurs="unbounded"/>
      <group ref="wsfl:activityFlowGroup"/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
   <attribute name="serviceProviderType" type="Qname”/>
</complexType>

<group name="activityFlowGroup">
   <sequence>
      <element name="export" type="wsfl:exportType"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="activity" type="wsfl:activityType"
               minOccurs=”0” maxOccurs="unbounded"/>
      <element name="controlLink" type="wsfl:controlLinkType"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="dataLink" type="wsfl:dataLinkType"
```

```
                minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
</group>
```

The <flowSource> element allows defining the message that provides the data when creating an instance of the model (its “initial context”); the flow source passes an “output” message that is typed by the input message of the flow model. The (output of a) flowSource can be linked to (input of) activities in the flow through dataLink attributes to indicate that those activities use flow model input data to perform their tasks. The <flowSink> element allows specification of the data that is returned once the instance is completed, and fault messages that are returned in case of an erroneous termination of an instance; the flow sink has an “input” message that is typed by the output message of the flow model. The (input of a) flowSink can be linked to (output of) activities in the flow model through dataLinks to indicate that the result of an activity contributes to the result of the overall flow model.

```
<complexType name="flowSourceType">
   <sequence>
      <element ref=”wsdl:output”/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
</complexType>
<complexType name="flowSinkType">
   <sequence>
      <element ref=”wsdl:input”/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
</complexType>
```

4.4 Service Providers and Service Bindings

Service providers represent the endpoints that provide services in flow models and global models. Actual service providers are determined based on locators as some sort of “instances” of service provider types.

4.4.1 Service Provider Types

Conceptually, a *service provider type* defines the external interface of some “type of service” (for example, bookseller, travel agent). It is different from a WSDL service, which describes a particular instance of some service (for example, the bookseller muchToRead.com, the travel agent getYouThere.co.uk). It is different from a WSDL port type, which describes the type of a WSDL port and which in turn, is bound to a single endpoint address.

The external interface (in the WSDL sense) of a service provider type is given by a *set* of WSDL port types. Port types can be defined “inline” within the service provider type; alternatively, a service provider type can import port types defined elsewhere. Some of these port types can be associated with WSFL flow models (which define dependencies between the operations), but this is not explicitly described in the service provider type. The following code is an example of a service provider type definition; this definition declares the single port type supported “inline” in contrast to referencing an “external” port type.

```
<serviceProviderType name=“bookseller”>
   <portType name=“processOrders”>
      <operation name=“receiveOrder”>
         <input message=“bookOrder”/>
      </operation>
      <operation name=“sendBooks”>
         <output message=“bookDelivery”/>
```

```
      </operation>
   </portType>
</serviceProviderType>
```

External portTypes are incorporated into the definition of a service provider type by means of an `<import>` element. The `portType` attribute is used to provide the qualified name of the portType being imported. The `<import>` element is also used to simplify the inline definition of portTypes. To this end, WSFL extends the syntax of the WSDL `<portType>` element to add the ability to reuse existing operation definitions through nested `<import>` elements. In this use of the `<import>` element, an `operation` attribute must also be provided to identify the operation definition being imported. See the following code for an example:

```
<portType name=”newPortType”>
   <import portType=”tns:somePortType”/>
   <import operation=”anOperation” portType=”tns:anotherPortType”/>
</portType>
```

The `<import>` element can be used in combination with the standard WSDL `<portType>` and `<operation>` elements. The schema syntax of the `<serviceProviderType>` is provided in the following code:

```
<complexType name=”serviceProviderTypeType”>
   <sequence>
      <element name=”portType” type=”wsfl:portTypeType”
               minOccurs=”0” maxOccurs=”unbounded”/>
      <element name="import"
               minOccurs="0" maxOccurs=”unbounded”>
         <complexType>
            <attribute name=”portType” type=”QName”/>
         </complexType>
      </element>
   </sequence>
</complexType>
```

The following code shows you the schema syntax for the WSFL `<portType>:`

```
<complexType name="portTypeType">
   <complexContent>
      <extension base="wsdl:portTypeType">
         <sequence>
            <element name="import"
                     minOccurs="0" maxOccurs=”unbounded”>
               <complexType>
                  <attribute name="portType" type="QName"/>
                  <attribute name="operation" type="NCName"/>
               </complexType>
            </element>
         </sequence>
      </extension>
   </complexContent>
</complexType>
```

4.4.2 Service Providers

Service providers are named and typed. The type is defined by referring to a service provider type. The service provider type states the functionality and behavior that the service provider is expected to provide in the flow model being defined. Service providers in a flow model are thus similar to typed variables, because they can be bound to any Web Service

satisfying the typing requirements. It is important to note that once a service provider is bound to a particular Web Service, all activities supported by operations of that provider get assigned to the specific service endpoints. A specific activity can sometimes overwrite this “default” binding by coding an activity locator element, see Section 4.5.1 “Activities.”

```
<serviceProvider name="mySupplier" type="dtp:supplier"/>
```

Service provider types are instantiated as service providers in flow models or in global models. A service provider is determined by a locator based on a service provider type. This may be obvious, but it should be noted that a service provider type is bound “as a whole” by a locator. Essentially, the locator identifies a particular WSDL service that implements the interface defined by the service provider type. See the following example:

```
<serviceProvider name=“bookseller01” type=“bookseller”>
   <locator type=“static” service=“muchToRead.com”/>
</serviceProvider>
```

The `<locator>` element is described in Section 4.4.3 “Service Locators.” The schema syntax for the `<serviceProvider>` element is given in the following example. The `<export>` element is described in Section 4.6 “Defining the Interface of a Flow Model,” and is discussed there in detail. It is also used within the context of global models (see Section 4.8 “Global Model”).

```
<complexType name="serviceProviderType">
   <sequence>
      <element name="locator" type="wsfl:locatorType"
               minOccurs="0"/>
      <element name="export" type="wsfl:exportType"
               minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
   <attribute name="type" type="QName" use=”required”/>
</complexType>
```

4.4.3 Service Locators

Service providers are the units of the binding operation in flow models. The assignment of services to service provider variables is controlled with the optional `<locator>` element, which has to return exactly one service. If present, the `<locator>` element is nested inside the `<serviceProvider>` element. If not present, the `<locator>` element that is nested inside the plug link of the activity to be started is taken. In this case, the middleware evaluating the plug link construct must return the value for the service chosen to allow the flow engine to set the value for the `<serviceProvider>` element.

Several different types of binding are possible, which are identified using the `type` attribute in the `<locator>` element. The syntax of the locator element changes with the locator type:

- In a **static** binding, the actual bound service is directly specified as the value of the `service` attribute (of type QName), referring to the WSDL or WSFL definition of the service:

  ```
  <locator type=”static” service=”tns:service1”/>
  ```

- In a **local** binding, the provider of the service is specified as a locally accessible program or software component (Java™ class, database-stored procedure, and so on). The `service` attribute can be used to specify the local service if a WSDL binding is

available for the local component. The `local` keyword is then used as a hint to the processor about the local nature of the service. However, to simplify the use of local bindings and support legacy-type specifications, the `<locator>` element can instead contain extensibility elements that specify how to bind to the local provider.

•

```
<locator type=”local”>
  <!-- extensibility elements                                    -->
</locator>
```

- In a UDDI type binding, the locator contains a UDDI query that will produce a list of candidate bindings when executed against a UDDI repository. Because the UDDI query application programming interface (API) is defined as a SOAP API, an UDDI query is represented by the corresponding SOAP message. This XML fragment is provided directly nested under the `<locator>` element:

```
<locator type=”uddi”
         bindTime=”startup”
         selectionPolicy=”first”>
   <uddi-api:find_service
            businessKey="uuid_key"
            generic="1.0"
            xmlns:uddi-api="urn:uddi-org:api">
      ...
   </uddi-api:find_service>
</locator>
```

The example above shows the nested UDDI query under the `<locator>` element. It also includes two attributes, `bindTime` and `selectionPolicy`, that help control the binding process. The `bindTime` attribute is used to specify at what point in time the service provider is to be bound:

- The value is set to `startup` to indicate that the UDDI query is to be performed at the time the flow model is instantiated.
- It is set to `firstHit` to indicate that the UDDI resolution must occur the first time an operation (activity) provided by that service provider is activated.
- It is set to `deployment` to indicate that the service is determined at the time the flow model is bound to a particular environment.

The `selectionPolicy` attributes determine how to choose a provider if the UDDI query returns more than one matching provider. There are currently three possible values of this attribute:

- `first` indicates that the first provider in the returned list should be selected.
- `random` indicates a random pick from the list.
- `user-defined` indicates that the user is providing a selection procedure, whose name is then encoded as the value of the `invoke` attribute. This case is shown in the next example:

```
<locator type=”uddi”
         bindTime=”firstHit”
         selectionPolicy=”user-defined”
         invoke=”pickUddi.exe”/>

  <!-- extensibility elements                                    -->
  <!-- e.g. to describe where the “invoke” procedure can         -->
  <!-- be found                                                  -->
```

```
   <uddi-api:find_service...> ... </uddi-api:find_service>
</locator>
```

- A locator can specify a value of `any` for the type attribute, indicating that the flow model does not place any restriction on what service provides the role of this service provider. This a convenient way to represent the situation in which the interaction with the `flowModel` is initiated by a third party, which at that point binds on to the role represented in the model by the service provider.

```
<locator type="any"/>
```

- In the **mobility** type binding, the information required to bind a service provider is extracted from the data exchanged in a prior interaction. For instance, a travel agency may select an airline on behalf of its customer and then send the airline contact information to the customer to enable her/him to deal with the airline directly. It must be assumed that the interaction carrying the binding information will be completed before the first activity supported by the mobility-type service provider is activated. A **default** binding may be provided with this locator type to avoid this situation.

  The mobility locator must identify the data field containing the binding information. For this purpose, attributes are provided to specify the name of the activity where the binding information was obtained, as well as the names of the input, output or fault message, of the message part, and of the data field where the information is contained:

```
<locator type="mobility"
         activity="getFlight"
         message="flightInfo"
         messagePart="airline"
         dataField="providerInfo">
```

  To support the aggregation model of Web Services, mobility locators are also supported as nested elements of plug links (see Section 4.7 “Plug Links”). Thus, at the aggregation level, the mobility binding information is contained in the input, output, or fault message of an operation of a port type of a service provider type.

```
<locator type="mobility"
         serviceProviderType="travelAgent"
         portType="bookings"
         operation="sendFlightInfo"
         message="flightInfo"
         messagePart="airline"
         dataField="providerInfo">
```

The schema syntax for the locator element is given below. The syntax below does not fully represent the variability of the `<locator>` element, because the schema definition language does not provide support for selecting alternate sets of attributes based on the value of one of them.

```
<complexType name="locatorType">
   <sequence>
      <any namespace="##other"
           minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="type" type="wsfl:locatorTypeType" />
   <attribute name="service" type="QName"/>
   <attribute name="bindTime" type="wsfl:bindTimeType"/>
   <attribute name="selectionPolicy"
              type="wsfl:selectionPolicyType"/>
```

```
   <attribute name="activity" type="NCName"/>
   <attribute name="message" type="NCName"/>
   <attribute name="messagePart" type="NCName" />
   <attribute name="dataField" type="string"/>
   <attribute name="serviceProviderType" type="QName"/>
   <attribute name="portType" type="QName"/>
   <attribute name="operation" type="NCName" />
   <attribute name="default" type="QName"/>
   <attribute name=”invoke” type=”string”/>
</complexType>
```

4.5 Defining Business Processes

The wiring of services into a business process (represented by a flow model) is described in WSFL using four kinds of elements: activities, control links, data links, and plug links. The metamodel corresponds to a special kind of a directed acyclic graph as described in Section 3.1 “Flow Metamodel.”

The nodes of the graph correspond to activities. An “activity” represents the use of an operation within the context of a flow. Through lifecycle operations, a whole (existing) flow model of a service can be used. When a whole flow model is used by an activity, the corresponding flow is often referred to as *subflow*.

Three types of edges are possible in the graph. “Control links” define the potential invocation sequence of activities in the model. “Data links” describe the flow of data between activities. Finally, “plug links” represent the explicit invocation of an operation offered by a different service provider as implementation of an activity. If the activity uses another flow, the operations of the corresponding lifecycle interface (see Section 3.2 “Lifecycle Interface”) are exploited for plugging.

Although arbitrary cycles are not allowed in the metamodel, it is possible to iterate activities, as explained in the next section. Because activities can use other flows, even flows can be iterated.

4.5.1 Activities

Activities are represented by `<activity>` elements. Activities are named, and have a signature that specifies the required inputs, outputs and the possible faults of the execution. The signature of an activity is specified in the same manner as the signature of a WSDL operation, using nested input, output, and fault elements, see [1].

```
<activity name="receiveTheMoney">
   <input name=”receiveInput” message=”tns:payForm”/>
   <output name=”acknowledgement” message=”tns:receiptForm”/>
</activity>
```

When the `name` attribute is not specified in an `<input>` or an `<output>` element, the `name` attribute defaults to the activity name with the suffix “Input” or “Output” appended, respectively. The name attribute is mandatory for `<fault>` elements.

4.5.1.1 Activity Implementation

An activity is executed by interacting with an operation from the interface of a service provider. This service provider is specified by a nested `<performedBy>` element. The implementing operation is specified through a nested `<implement>` element. If the implementation of the operation can be provided “internally,” requiring no access to an “external” service provider, the `<implement>` element specifies this by using a nested

<internal> element (see Section 6 “Appendix B: Internal Activity Implementations“ for sample extensibility elements of the WSDL definitions of the port types required in this case to bind to local implementations like EXE, DLL, or other executables).

Otherwise, a nested <export> element is used. Both the <internal> element as well as the <export> element specify the operation that is “dual” to the one being the final target implementation; a plug link (see Section 4.7 “Plug Links”) is used to connect the dual operation with the final target implementation. The <export> element specifies the dual operation by referencing an operation from the flow’s public interface. These two elements are described in more detail in Section 4.6 “Defining the Interface of a Flow Model.”

```
<flowModel name=“bookLover” serviceProviderType=“bookLoverPublic”>

   <activity name=“selectBook”>
      <performedBy serviceProvider=“local”/>
      <implement>
         <internal serviceProviderType=“bookLoverPrivate”
                   portType=“bookStuff” operation=“selectBook”/>
      </implement>
   </activity>

   <activity name=“orderDictionary”>
      <input message=“bookOrder”/>
      <performedBy serviceProvider=“bookseller01”/>
      <implement>
         <export>
            <target portType=“bookRequester”
                    operation=“orderDictionary”/>
         </export>
      </implement>
   </activity>

</flowModel>
```

Note that the operation implementing an activity can be a lifecycle operation used to control the instance of another flow. Depending on the lifecycle operation used to plug a subflow, the execution of the flow model can take place either as part of the execution of the current flow, or can be spawned as a new independent flow.

> V1.0 remark: The signatures of an activity and the signature of its implementing operation need not match in general. Nested <map> elements (see Section 4.5.3 “Data Links and Data Mapping”) would then be used to relate the inputs, outputs and faults of the two. WSFL Version 1.0, however, assumes that the two signatures match, thus making the mapping unnecessary.

In order to determine the actual endpoint that provides the implementation proper (that is, not a “proxy”) of the activity, a locator is used. If the service provider associated with the activity has no locator assigned, the locator of the plug link associated with the activity is evaluated considering the assigned service provider. Especially in this case, it is convenient to specify the corresponding plug link directly with the activity.

```
<activity name=“orderDictionary”>
   <implement>
      <export>
         <target portType=“bookRequester”
                 operation=“orderDictionary”/>
         <plugLink>
            <target portType=”sellBooks”
                    operation=”receiveOrder”/>
```

```
            <locator type=”mobility”
                     activity=”selectBook”
                     message=”tns:comma123”
                     messagePart=”sellerChosen”
                     default=”tns:mySeller”>
         </plugLink>
      </export>
   </implement>
</activity>
```

4.5.1.2 Exit Condition

The successful completion of an activity leads to the possible activation of new activities, that is, activities that succeed the successfully completed activity through control links. It also leads to the propagation of data according to the data links starting at the activity. An “exit condition” can be imposed on the activity to control whether it completed successfully. This is represented by the `exitCondition` attribute, whose value is a Boolean expression relating values of inputs and outputs of the activity itself or of other preceding activities. The activity is considered to have completed successfully only if the exit condition evaluates to true. The `exitCondition` attribute is optional.

The exit condition serves a second purpose: Some activities may require that their execution be repeated until a certain condition is met. The exit condition is used to provide a Boolean condition for the termination of the iterative execution. The operation or flow model associated with the activity will be executed at least once, and as many times as needed until the Boolean expression evaluates to true. By this mechanism, a controlled kind of cycles (that is, do-until loops) is represented in a flow.

```
<activity name="receiveTheMoney"
          exitCondition=”receiptForm.status='OK'”/>
```

> V1.0 remark: A future version of WSFL might support a `doUntil` attribute of the `<activity>` element. This will allow a clean separation of the specification of the looping condition and the condition that measures the successful completion of each iterated execution of an activity.

4.5.1.3 Join Condition

The execution of parallel branches in a flow can be synchronized at “join activities,” that is, activities that are the target of more than one control link (see Section 3.1.1 “Syntax”). Join activities control the synchronization of parallel branches through “join conditions” that are associated with join activities. A join condition is a Boolean expression in the names of the control links (see Section 4.5.2 “Control Links”). The nested `<join>` element is used to specify the value of this Boolean expression as its `condition` attribute.

A join condition is evaluated by substituting the names of the control links of the expression by the truth-values of the transition conditions of the referenced control links. The `when` attribute of the join condition allows to specify the point in time at which the join condition is evaluated: A value of `deferred` requires to wait until the transition conditions of all referenced control links have been evaluated; this is the default value. A value of `immediate` requires evaluating the join condition whenever a transition condition of a reference control link has been evaluated. The truth-value of an immediately evaluated join condition is considered to be final as soon as it is known that the truth-value of the condition can no longer change.

```
<activity name="receiveTheMoney">
   <join condition=”POaccepted AND SRreceived” when=”deferred”>
</activity>
```

V1.0 remark: `deferred` is the only valid value for the `when` attribute in Version 1.0.

4.5.1.4 Container Materialization

The construction of the input message of an activity can be defined through data links (see Section 4.5.3 “Data Links and Data Mapping”). Because an activity can be the target of multiple data links, a mechanism for resolving conflicting data mapping specifications is needed. The nested `<materialize>` element is used for this purpose: It either contains a `<mapPolicy>` element or a `<construction>` element.

A map policy specifies through its `order` attribute the order in which the data maps have to be applied. Possible values include:

- `LWW` - the maps have to be applied in their “last writer wins” order (this value is the default)
- `FWR` - the maps have to be applied in their “first writer wins” order
- `RANDOM` - the maps are applied in a random order

If none of the predefined values is specified, the value of the order attribute is a list of blank-separated names of data links; the map elements of the data links will be applied in exactly the specified order.

```
<activity name="receiveTheMoney">
   <materialize>
      <mapPolicy order=”LWW”/>
   </materialize>
</activity>
```

A construction specifies a particular manner how the container must be materialized: The `type` attribute defines the technology used to construct the message; a value of `XSLT` is its default. The `location` attribute specifies where the construction prescription can be found.

```
<activity name="acceptShipmentRequest">
   <materialize>
      <construction type=”XSLT”
                    location=”mapRepository.com/x213.xsd”/>
   </materialize>
</activity>
```

V1.0 remark: Materialization of containers through explicit construction elements is not subject of WSFL Version 1.

4.5.1.5 Summary: Activity Schema

The schema syntax for the activity element is provided in the following example:

```
<complexType name="activityType">
   <complexContent>
      <extension base="wsdl:operationType">
         <sequence>
            <element name=”performedBy”>
               <complexType>
                  <attribute name=”serviceProvider”
                             type=”NCName”/>
               </complexType>
            </element>
            <element name="implement">
               <complexType>
                  <choice>
                     <element name="internal"
```

```
                              type=”wsfl:internalType”/>
                     <element name="export"
                              type="wsfl:exportType"/>
                  </choice>
               </complexType>
            </element>
            <element name="join" type="wsfl:joinType"
                     minOccurs="0"/>
            <element name="materialize" type="wsfl:materializeType"
                     minOccurs="0"/>
            <any namespace="##other" minOccurs="0"
                 maxOccurs="unbounded"/>
         </sequence>
         <attribute name="name" type="NCName"/>
         <attribute name="exitCondition" type="string"/>
      </extension>
   </complexContent>
</complexType>

<complexType name="joinType">
   <attribute name="condition" type="wsfl:NCNameList"
                    use="required"/>
   <attribute name="when" type="wsfl:whenType" use="default"
              value="deferred"/>
</complexType>

<complexType name="materializeType">
   <choice>
      <element name="mapPolicy">
         <complexType>
            <attribute name="order" type="wsfl:orderType"
                       use="default" value="LWW"/>
         </complexType>
      </element>
      <element name="construction">
         <complexType>
            <attribute name="type" type="string"
                       use="default" value="XSLT"/>
            <attribute name="location" type="string"/>
         </complexType>
      </element>
   </choice>
</complexType>
```

4.5.2 Control Links

Control links are used to define the control flow among the activities of the model. A control link describes an activity (the “source”) and its possible successor activities (the “targets”). The `<controlLink>` element is used to represent control links, and the mandatory `source` and `target` attributes are used to name the linked activities. The two linked activities must have been defined within the flow model that contains the definition of the control link. Observe that we do not connect the operations that are used by the activities as their implementations, but the activities themselves. This is because an operation may appear as implementation of more than one activity.

Control links can carry conditions that are used as “guards” for following the potential path from the source to the target (“transition conditions”). The actual truth-value of the transition condition determines at execution time whether or not the corresponding link is followed and the target activity is considered for activation. The Boolean expression of the transition

condition is provided as the value of the `transitionCondition` attribute. Refer to Section 3.1.2 “Operational Semantics” for what happens when the transition condition evaluates to false (“dead path elimination”).

The Boolean expression representing a transition condition is an expression in a subset of the data fields of any part of the input, output, or fault messages of activities preceding the source activity (including the latter activity itself) of the control link. See Section 4.5.3 “Data Links and Data Mapping” for an explanation of how to identify and use data field values. The encoding of this Boolean expression depends on the type system that was used to define the data types of the messages and their parts. In the case where XML Schema is used, the Boolean expression will be an XPath expression.

For example:

```
<controlLink source="processPO"
             target="acceptSR"
             transitionCondition=
                      ”processPOOutput/x &gt; acceptSRInput.y”/>
```

Recall that activities have WSDL signatures, so different kinds of “results” are possible: either a regular output message or one of its fault messages will be returned. A control link may refer to the particular kind of the output using the `result` attribute, which identifies the corresponding output by name (for example, to identify a particular fault message). The control link will be followed if, and only if, the message of the specified name is returned.

In particular, this mechanism provides a straightforward way for exception handling: Besides specifying the regular flow between activities flow modelers can specify control links from an activity to “error handling” activities. Which of the error handling activities are to be run is controlled by an appropriate value of the `result` attribute, specifying the name of the fault message returned by the source activity of the control link.

For example:

```
<controlLink name=”sup-ship-1”
             source="processPO"
             target="acceptSR"
             result=”invalidInputMessage”/>
```

Control links do have an optional name. This name must be provided in order to refer to a control link in a join condition.

The schema syntax for the `<controlLink>` element is as follows:

```
<complexType name="linkType">
   <attribute name="name" type="NCName"/>
   <attribute name="source" type="NCName"/>
   <attribute name="target" type="NCName"/>
</complexType>

<complexType name="controlLinkType">
   <complexContent>
      <extension base="linkType">
         <attribute name="transitionCondition" type="string"/>
         <attribute name="result" type="NCName"/>
      </extension>
   </complexContent>
</complexType>
```

4.5.3 Data Links and Data Mapping

Data links define the exchange of information between activities, and it is represented by the <dataLink> element. The attributes source and target specify the linked activities (or the flow source or flow sink of the overall flow, respectively. See Section 4.3 “Flow Models”). The information exchanged can originate in any part of the input, output or failure messages of the source activity, and is mapped to any part of the input messages of the target activity. It is a requirement that a control path (a continuous path made up of control links) exist from the source activity to the target activity (see Section 3.1.1 “Syntax”). The specific mapping between source and target data elements is defined using a nested <map> element:

```
<dataLink name="sup-shp1"
          source="processPO"
          target="acceptRequest">
   <map sourceMessage="anINVandSR" targetMessage="anSR"
        sourcePart=”SR” targetPart=”SR”/>
</dataLink>
```

The <map> element provides a general mechanism for specifying data mapping and data conversion in WSFL (see also Sections 3.1.1.12 “Data Links,” 4.5.3 “Data Links and Data Mapping,” 4.6.3 “Exporting Activities,” 4.6.4 “Exporting Operations” and 4.7 “Plug Links”). The <map> element can appear nested inside a <dataLink>, <export>, or <plugLink> element; the enclosing element identifies a source and a target activity or operation whose signatures define the data elements being mapped.

The messages that contain the source and target data are specified using the sourceMessage and targetMessage attributes, whose value must be the name of an <input>, <output> or <fault> element. Recall that WSDL provides a default name for the <input> and <output> elements of an operation definition, and requires that a name be provided for all <fault> elements. A similar convention is used in the definition of the signature of an activity (see Section 4.5.1 “Activities”).

Four additional attributes are available to determine the specific data elements that are being mapped. The sourcePart and targetPart attributes determine the WSDL part of the messages that contain the mapped data, while the sourceField and targetField attributes are used to point a specific field in the message part. The value encoded in the last attributes depends on the type system that was used to define the datatype of the part. If the XML Schema is used, the value is an XPath expression. Finally, a converter attribute can be used to specify a user-provided function that performs the data mapping and conversion, for example for XSLT processing.

> V1.0 remark: The converter attribute is not supported in WSFL Version 1.

In order to map the flow model input to the input of an activity or to map output of an activity to the flow model output, the <flowSource> and <flowSink> elements of a flow model are used as sources and targets of data links. As explained above, flow source and flow sink are named elements in a flow model, which have an output and respectively, an input message defining the external interface of a flow model. Just as data links connect outputs of activities with inputs of other activities, they can be used to connect the output of the <flowSource> element to the input of activities, and the output of activities to the input of the <flowSink> element.

Mapping data from the flow model input to an activity can then be done as follows (assuming that the flow source in the flow model has the name flowSource):

```
<dataLink name="flowModel-ship"
          source="flowSource"
```

```
          target="acceptRequest">
   <map sourceMessage="anINVandSR" targetMessage="anSR"
        sourcePart=”SR” targetPart=”SR”/>
</dataLink>
```

Mapping data from an activity to the flow model output can be done as follows (assuming that the flow sink in the flow model has the name `flowSink`):

```
<dataLink name="ship-flowmodel"
          source="processPO"
          target="flowSink">
   <map sourceMessage="anINVandSR" targetMessage="anSR"
        sourcePart=”SR” targetPart=”SR”/>
</dataLink>
```

The schema syntax for the `<map>` and `<dataLink>` elements is given in the following code:

```
<element name="map">
   <complexType>
      <attribute name="sourceMessage" type="NCName"/>
      <attribute name="targetMessage" type="NCName"/>
      <attribute name="sourcePart" type="NCName"/>
      <attribute name="targetPart" type="NCName"/>
      <attribute name="sourceField" type="NCName"/>
      <attribute name="targetField" type="NCName"/>
      <attribute name="converter" type="string"/>
   </complexType>
</element>
```

```
<complexType name="dataLinkType">
   <complexContent>
      <extension base="linkType">
         <sequence>
            <element ref="map" minOccurs="0"/>
         </sequence>
      </extension>
   </complexContent>
</complexType>
```

When a data link does not contain a `<map>` element, it is assumed that the output message of the source activity is identically passed as input message to the target activity.

The name of a data link is optional. When an activity is the target of more than one data link, these data links are named to allow an easy specification of the order in which possibly conflicting data links are to be applied.

4.6 Defining the Interface of a Flow Model

The interface of a flow model includes two groups of operations: the set of operations and port types that the model makes available for third-party interaction constitute the *external* or *public* interface of the model. The external interface of a flow model always includes the lifecycle interface of the flow model, described in Sections 3.2 “Lifecycle Interface“ and 4.6.5 “Support for Lifecycle Operations,” in addition to any other supported port types. The set of operations and port types that the flow model requires to interact with internal services (for example, application components, TP monitors, and so on) constitute its *internal* or *private* interface; all the operations of the internal interface are, by definition, plug-linked to internal providers, and are not available for interaction with other services.

The following sections describe the differences between the public and private interfaces, and the mechanisms available in WSFL for defining the interface of a flow model: exporting activities, exporting operations from internal service providers, and exporting lifecycle operations.

4.6.1 External and Internal Interfaces

The operations of the external or public interface of a flow model define how the model interacts with external partners; both the functional capabilities as well as the requirements against partners are specified by the public interface. All the public operations must be included in one of the supported port types declared using the `service provider type` attribute on the `<flowModel>` element. (see Section 4.3 “Flow Models”). This attribute provides the name of a service provider type that in turn denotes a set of qualified names that refer to port types defined in WSDL or WSFL documents.

The operations in the internal or private interface of a flow model are those required to support the interaction between the flow and the internal service providers used in its definition. Because these providers are in many cases remote, they are accessed through plug links (remote invocations) and thus require supporting endpoints in the interface of the flow. Because these endpoints are already bound and are not available to external partners, they are not declared in the service provider type attribute of the flow model. They relate to the definition and implementation of the flow model, not its ability to interact with external partners.

In both cases, however, the specific effect of an interaction must be defined by mapping each operation to the internal behavior of the model. The mapping refers capabilities and requirements back into the flow and reveals the behavior (for example, constraints) of the whole set of operations. The `<export>` element is used in WSFL to describe the mapping between activities, lifecycle operations, or service provider operations, and operations of the public interface of a flow model. The `<internal>` element is used to map between activities and operations of the private interface. The precise semantics of the export operation are described in Section 3.5.1 “Composition Metamodel Overview” and 3.5.6 “Exporting Operations.”

The schema type representing all the uses of the `<export>` element is given in the following example, and is followed by the type for the `<internal>` element:

```
<complexType name="exportType">
   <sequence>
      <element name=”source” type=”endPointType”
               minOccurs=”0”/>
      <element name=”target” type=”endPointType”/>
      <element ref="wsfl:map"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name=”plugLink” type=”wsfl:plugLinkType”
               minOccurs="0"/>
   </sequence>
   <attribute name="lifecycleAction" type="NCName"/>
</complexType>

<complexType name=”internalType”>
   <complexContent>
      <extension base="wsfl:endPointType"/>
         <sequence>
            <element name=”plugLink” type=”wsfl:plugLinkType”/>
         </sequence>
      </extension>
```

```
   </complexContent>
</complexType>

<complexType name="endPointType">
   <attribute name="serviceProvider" type="NCName"/>
   <attribute name="serviceProviderType" type="NCName"/>
   <attribute name="portType" type="QName" use=”required”/>
   <attribute name="operation" type="NCName" use=”required”/>
</complexType>
```

4.6.2 Internal Implementations

The <internal> element is used to map activities to implementing operations that are accessed through the internal interface of a flow model. The <internal> element appears always nested inside an <implement> element in the activity definition (see Section 4.5.1.1 “Activity Implementation”). The operation on the internal interface is identified by the service provider type of the internal interface, the port type and the operation. Note that the type of the internal interface is not part of the type of the flow model, because only public interface operations are included there, and declared by the serviceProviderType attribute of the <flowModel> element.

The <internal> element can contain a nested plug link that connects the internal operation to the actual provider. An example was provided in Section 4.5.1.1 “Activity Implementation.”

4.6.3 Exporting Activities

The mapping of an activity to an operation of the public interface of a flow model is represented in WSFL by an <export> element. The corresponding export element is included in the definition of the implementation of the activity, that is, its associated <implement> element (see Section 4.5.1.1 “Activity Implementation”). The <export> element only needs to specify a <target> nested element that identifies the target operation on the public interface.

If the signature of the target operation does not match the signature of the source activity or lifecycle operation, one or more <map> elements (described in Section 4.5.3 “Data Links and Data Mapping”) can be provided to define the mapping between input and output messages of the external operation and the activity or lifecycle operation.

```
<activity name =”processPO”>
   <implement>
      <export>
         <target portType="tts:totalSupplyPT"
                 operation="sendOrder"/>
            <map sourceMessage=”aPOandSR”
                 targetMessage=”aPO”
                 sourcePart="PO"
                 targetPart="PO"/>
      </export>
   </implementation>
</activity>
```

When the activity is implemented by an operation of the internal interface of the model, the <internal> element has to be used.

V1.0 remark: Multiple activities of a particular flow model may be exported to the same operation of the public interface. WSFL Version 1.0, however, does not make use of this capability.

4.6.4 Exporting Operations

Within global models, it is possible to export an operation of an encompassed service provider, defining the implementation of an operation in the public interface of the global model as simple delegation to the encompassed service provider.

The `<export>` element is in this case nested inside the `<serviceProvider>` element to which the implementation is delegated. The `<source>` element specifies the port type and operation providing the implementation, and the `<target>` element specifies the operation of the public interface whose implementation is delegated to the source. As before, `<map>` elements can be used to specify the required adaptation of non-matching signatures. In the example, the `buy` operation of the `globalPortType` port type is delegated to the `spawn` operation of the `lifeCycle` port type of the `bookLover` service provider.

```
<serviceProvider name=“bookLover” type=“bookLoverPublic”>
   <export>
      <source portType=“lifeCycle” operation=“spawn”/>
      <target portType=“globalPortType” operation=“buy”/>
   </export>
</serviceProvider>
```

4.6.5 Support for Lifecycle Operations

Managing the lifecycle of a flow is done by lifecycle operations. WSFL defines a set of lifecycle operations that can be supported by a flow model. The external interface of a lifecycle operation is defined by an `<export>` as well. It appears in this case as a direct child of the `<flowModel>` element, and uses the `lifecycleAction` attribute to name the operation to be exported. A nested `<target>` element identifies the operation on the public interface that is mapped to the lifecycle action.

In the following example, the lifecycle operation `call` is exported as operation `CreditRequest` of port type `CreditRequestService`. The message of the exported operation (`aCreditRequest`)is mapped to the input message of the lifecycle operation (`aCR`) by specifying that the corresponding `Person` part be copied.

```
<export lifecycleAction="call">
   <target portType="tts:creditRequestService"
           operation="CreditRequest"/>
   <map sourceMessage=”aCreditRequest” targetMessage=”aCR”
        sourcePart="Person" targetPart="Person"/>
</export>
```

The port types that can be used to export a lifecycle operation to can be any one in the service provider type of the flow model being defined. Data mapping between the signature of the port type operation and the signature of the lifecycle operation can be described using `<map>` elements as before. For one lifecycle operation, several port type operations can be defined, for example, to support the same lifecycle operation but with different signatures.

WSFL defines a set of lifecycle operations. All operations are request-response operations, that is, they receive a message and send a correlated message. In order to support a lifecycle operation in a flow model, the operation has to be defined by the `<export>` element

explicitly. The signature of the lifecycle operation is implicit defined (see the following paragraph). Each service provider that supports a flow model provides a port definition for every lifecycle operation by which the operation is bound to a binding and an address.

In the following sections, the WSFL lifecycle operations are described in detail. We use WSDL to describe their signature. The placeholder name of the flow model input data message that is used in the flow model's signature is `msgIn`, and the name representing the flow model output data is `msgOut`. Note that the signature of the lifecycle operations is implicitly defined, that is, derived from the `<flowSource>` and `<flowSink>` elements (see Section 4.3 “Flow Models”), and does not need to be defined explicitly; it is presented in the following sections only to explain the signature of the lifecycle operations.

4.6.5.1 Lifecycle Operation `spawn`

The lifecycle operation `spawn` creates an instance of the flow model and starts it. As an immediate result of the start operation, a unique instance identifier called `FlowInstanceID` is returned in the `SpawnResult` message; optionally, an implementation might also return the time at which the instance has been created.

```
<operation name="spawn">
   <input message="msgIn">
   <output message="wsfl:SpawnResult">
   <fault message="wsfl:Fault">
</operation>

<complexType name="SpawnResult">
   <sequence>
      <element ref="wsfl:FlowInstanceID"/>
      <element ref="wsfl:FlowInstanceCreationTime"
               minOccurs="0" maxOccurs="1"/>
   </sequence>
</complexType>

<element name="FlowInstanceID" type="string"/>
<element name="FlowInstanceCreationTime" type="dateTime"/>
```

The `<fault>` element provides more details about the reason why the operation did not succeed.

```
<complexType name="Fault">
   <sequence>
      <element name="MainCode" type="integer"/>
      <element name="SubCode" type="integer"
               minOccurs="0" maxOccurs="1"/>
      <element name="MessageText" type="string"
               minOccurs="0" maxOccurs="1"/>
   </sequence>
</complexType>
```

The `MainCode` attribute is an operation-specific error code that indicates what went wrong. The `SubCode` details the main fault code; for example, when the main code specifies `Invalid Key`, the sub-code could tell that the format passed is wrong. The `MessageText` is a description about the error that occurred. Some `spawn` specific `MainCode` values are listed below.

Faults:

- WSFL_ERROR_FLOW_MODEL_DOES_NOT_EXIST - The specified flow model does not exist. The flow model is not known by the flow model engine and therefore cannot be instantiated.
- WSFL_ERROR_INVALID_INPUT_MESSAGE - The input message does not conform to the input message of the flow model
- WSFL_ERROR_OPERATION_FAILED - The operation cannot be accomplished because of some temporary internal error in the flow model engine

4.6.5.2 Lifecycle Operation call

The lifecycle operation call executes an instance of the flow model. The flow model output message is returned when the flow model instance completes.

```
<operation name="call">
   <input message="msgIn">
   <output message="msgOut">
   <fault message="wsfl:Fault">
</operation>
```

Faults:

- WSFL_ERROR_FLOW_MODEL_DOES_NOT_EXIST - The specified flow model does not exist. The flow model is not known by the flow model engine and therefore cannot be instantiated.
- WSFL_ERROR_INVALID_INPUT_MESSAGE - The input message does not conform to the input message of the flow model.
- WSFL_ERROR_OPERATION_FAILED - The operation cannot be accomplished because of some temporary internal error in the flow model engine.

4.6.5.3 Lifecycle Operation enquire

The lifecycle operation enquire queries the status of the flow model instance; the instance identifier FlowInstanceID of the flow is the input of this lifecycle operation. As a result, the status of the flow model instance is returned. The EnquiryResult message consists of the instance identifier of the flow, its current state, the time the instance has been created, and the time the state of the instance has changed most recently. According to Section 3.3 “Business Process Lifecycle,” a flow instance can be in two different states, namely Running or Suspended.

```
<operation name="enquire">
   <input message="wsfl:EnquiryInput">
   <output message="wsfl:EnquiryResult">
   <fault message="wsfl:Fault">
</operation>

<complexType name="EnquiryInput">
   <sequence>
      <element ref="wsfl:FlowInstanceID"/>
   </sequence>
</complexType>

<complexType name="EnquiryResult">
   <sequence>
      <element ref="wsfl:FlowInstanceID"/>
      <element ref="wsfl:FlowInstanceState"/>
      <element ref="wsfl:FlowInstanceCreationTime"
               minOccurs="0" maxOccurs="1"/>
```

```
      <element ref="wsfl:FlowInstanceLastModificationTime"
               minOccurs="0" maxOccurs="1"/>
   </sequence>
</complexType>

<element name="FlowInstanceState" type="wsfl:state"/>
<element name="FlowInstanceLastModificationTime" type="dateTime"/>

<simpleType name="state" base="string">
   <enumeration value="Running"/>
   <enumeration value="Suspended"/>
</simpleType>
```

Faults:

- `WSFL_ERROR_INSTANCE_DOES_NOT_EXIST` - The instance ID provided in the input message refers to a flow model instance that does not exist
- `WSFL_ERROR_OPERATION_FAILED` - The operation cannot be accomplished because of some temporary internal error in the flow model engine

  V1.0 remark: In WSFL 1.0 the only valid input is a `FlowInstanceID`, and the status of the overall flow is reported back in `wsfl:EnquiryResult`. In the future, more detailed information, for example, the current activities running might be requested and reported back to facilitate flow model monitoring.

4.6.5.4 Lifecycle Operation `terminate`

The lifecycle operation `terminate` terminates the flow model instance.

```
<operation name="terminate">
   <input message="wsfl:FlowInstanceID">
   <output message="wsfl:Success">
   <fault message="wsfl:Fault">
</operation>

<simpleType name="Success" base="string">
   <enumeration value="OK"/>
</simpleType>
```

Faults:

- `WSFL_ERROR_INSTANCE_DOES_NOT_EXIST` - The instance ID provided in the input message refers to a flow model instance that does not exist.
- `WSFL_ERROR_INVALID_STATE_TRANSITION` - The state of the flow model instance does not allow this operation, for example, the flow model is in a suspended state.
- `WSFL_ERROR_OPERATION_FAILED` - The operation cannot be accomplished because of some temporary internal error in the flow model engine.

4.6.5.5 Lifecycle Operation `suspend`

The lifecycle operation `suspend` suspends the flow model instance.

```
<operation name="suspend">
   <input message="wsfl:FlowInstanceID">
   <output message="wsfl:Success">
   <fault message="wsfl:Fault">
</operation>
```

Faults:

- `WSFL_ERROR_INSTANCE_DOES_NOT_EXIST` - The instance ID provided in the input message refers to a flow model instance that does not exist.
- `WSFL_ERROR_INVALID_STATE_TRANSITION` - The state of the flow model instance does not allow this operation, for example, the flow model is in a suspended state.
- `WSFL_ERROR_OPERATION_FAILED` - The operation cannot be accomplished because of some temporary internal error in the flow model engine.

4.6.5.6 Lifecycle Operation resume

Lifecycle operation `suspend` suspends the flow model instance.

```
<operation name="resume">
   <input message="wsfl:FlowInstanceID">
   <output message="wsfl:Success">
   <fault message="wsfl:Fault">
</operation>
```

Faults:

- `WSFL_ERROR_INSTANCE_DOES_NOT_EXIST` - The instance ID provided in the input message refers to a flow model instance that does not exist.
- `WSFL_ERROR_INVALID_STATE_TRANSITION` - The state of the flow model instance does not allow this operation, for example, the flow model is in state running.
- `WSFL_ERROR_OPERATION_FAILED` - The operation cannot be accomplished because of some temporary internal error in the flow model engine.

4.6.6 Putting Things Together: The External Interface of a Flow

For each activity in a flow model, a WSDL operation is defined that provides the implementation of the activity. This operation defines the interface of a proxy that is used by the activity to interact with the actual realization of the business function represented by the activity. The actual realization is defined by another WSDL operation that is dual to the proxy (for example, notify is dual to one-way, and so on). The association of a proxy and the realization proper is defined using *plug links* (see Section 4.7 “Plug Links”).

The interface of a flow model is defined by a set of port types, that is, a service provider type. Each operation in this interface represents the view on an activity “from the outside.” This view is defined by a WSDL operation; in effect, the activity uses this proxy to interact with some external Web Service. Not all activities within a flow model use external Web Services and therefore, not all activities are represented in the interface of a flow model. For activities that use external services an *export* clause defines the association between the activity and an entry in the definition of the external interface of the flow model.

The following example describes the external, that is, public interface of the “book lover” flow model, defining the structure of those activities that interact with other Web Services. First, the lifecycle interface of the flow model is defined; for simplicity, only a single operation is included in this interface. Next, the port type `bookRequester` subsumes all interactions of the flow model with external Web Services. For simplicity, in this example, we will assume that all the messages and service provider types have been defined in the same document, and that the namespace prefix `tns` corresponds to all local definitions.

```
<serviceProviderType name=“bookLoverPublic”>

   <portType name=“lifeCycle”>
      <operation name=“spawn”>
         <input message=“tns:budget”/>
```

```
         <output message="tns:SpawnResult"/>
      </operation>
   </portType>

   <portType name="bookRequester">
      <operation name="orderDictionary">
         <output message="tns:bookOrder"/>
      </operation>
      <operation name="receiveDictionary">
         <input message="tns:bookDelivery"/>
      </operation>
      <operation name="orderPoetry">
         <output message="tns:bookOrder"/>
      </operation>
      <operation name="receivePoetry">
         <input message="tns:bookDelivery"/>
      </operation>
   </portType>

</serviceProviderType>
```

The following service provider type defines the operation of the internal activity (that is, those that do not use an external service) in the "book lover" flow model:

```
<serviceProviderType name="bookLoverPrivate">
   <portType name="bookStuff">
      <operation name="selectBook">
         <output message="tns:comma123"/>
         <input message="tns:comma123"/>
      </operation>
   </portType>
</serviceProviderType>
```

The operation representing the actual implementation (in our case, a CICS® transaction) of the internal activity is defined by a separate service provider type that is used by the flow model (and potentially others). Note that this implementation proper, that is, operation `T123,` is a request-response operation receiving an input and returning a response. The dual-proxy operation `selectBook` from the `bookStuff` port type is defined as a solicit-response operation, sending out a message and expecting a response.

```
<serviceProviderType name="someCICSTransactions">
   <portType name="cicsy">
      <operation name="T123">
         <input message="tns:comma123"/>
         <output message="tns:comma123"/>
      </operation>
   </portType>
</serviceProviderType>
```

Now we are ready to define the "book lover" flow model. Note the `serviceProviderType` attribute of the `flowModel` element that references the service provider type `bookLoverPublic` that defines the public interface. The service provider `bookseller01` is used to order a dictionary and the service provider `bookseller02` is used to order some poetry. We use static locators for these service providers to keep things simply, but any other locator type is valid. The `local` service provider provides the implementations of the "internal" operations.

The activity `selectBook` requires interaction with the `local` service provider for its realization; this is specified through a `<performedBy>` element referring to this service

provider. The actual (proxy) implementation is defined by an `<implement>` element. Because the interaction between the proxy and the implementation proper is completely hidden from the outside, an `<internal>` element is used to indicate this. Finally, the internal proxy operation is plug-linked to the corresponding operation of the `local` service provider; the associated *plug link* is defined "inline" here, but it could be done in a separate "global model."

The other activities use "external" services for their implementations. Thus, the associated `<performedBy>` element references an external service provider. The corresponding `<implement>` element contains an `<export>` element associating the implementing proxy with an operation within the flow's public interface. Plug linking is done in a separate global model, that is, not "inline" as done before for the internal implementation.

```
<flowModel name="bookLover" serviceProviderType="bookLoverPublic">

   <serviceProvider name="bookseller01" type="tns:bookseller">
      <locator type="static" service="muchToRead.com"/>
   </serviceProvider>

   <serviceProvider name="bookseller02" type="tns:bookseller">
      <locator type="static" service="allYouCanRead.com"/>
   </serviceProvider>

   <serviceProvider name="local" type="tns:someCICSTransactions"/>

   <activity name="selectBook">
      <input message=" tns:comma123"/>
      <output message=" tns:comma123"/>
      <performedBy serviceProvider="local"/>
      <implement>
         <internal serviceProviderType="tns:bookLoverPrivate"
                   portType="tns:bookStuff" operation="selectBook">
            <plugLink>
               <source serviceProviderType="tns:bookLoverPrivate"
                       portType="tns:bookStuff"
                       operation="selectBook"/>
               <target serviceProviderType="tns:someCicsTransactions"
                       portType="tns:cicsy" operation="T123"/>
            </plugLink>
         </internal>
      </implement>
   </activity>

   <activity name="orderDictionary">
      <input message="tns:bookOrder"/>
      <performedBy serviceProvider="bookseller01"/>
      <implement>
         <export>
            <target portType="tns:bookRequester"
                    operation="orderDictionary"/>
         </export>
      </implement>
   </activity>

   <activity name="orderPoetry">
      <input message="bookOrder"/>
      <performedBy serviceProvider="bookseller02"/>
      <implement>
         <export>
            <target portType="tns:bookRequester"
```

```
                    operation="orderPoetry"/>
         </export>
      </implement>
   </activity>

   <activity name="receiveDictionary">
      <output message="tns:bookDelivery"/>
      <performedBy serviceProvider="bookseller01"/>
      <implement>
         <export>
            <target portType="tns:bookRequester"
                    operation="receiveDictionary"/>
         </export>
      </implement>
   </activity>

   <activity name="receivePoetry">
      <output message="tns:bookDelivery"/>
      <performedBy serviceProvider="bookseller02"/>
      <implement>
         <export>
            <target portType="tns:bookRequester"
                    operation="receivePoetry"/>
         </export>
      </implement>
   </activity>

   <controlLink source="selectBook" target="orderDictionary"/>
   <controlLink source="orderDictionary" target="receiveDictionary"/>
   <controlLink source="selectBook" target="orderPoetry"/>
   <controlLink source="order Poetry" target="receivePoetry"/>

   <dataLink source="selectBook" target="orderDictionary"/>
   <dataLink source="selectBook" target="orderPoetry"/>

</flowModel>
```

4.7 Plug Links

Plug links are used in WSFL to model the interaction between remote service providers. A plug link represents in WSFL the invocation by one service provider of an operation of the public interface of another service provider. Unlike control and data links, plug links do not connect activities; rather, they connect two operations with "dual" signatures.

The source operation of a plug link (on the "calling" service provider) must have a signature corresponding to a "notification" or a "solicit-response" operation (as defined in the WSDL specification, [1]). This represents the ability of the caller to initiate the invocation request. Correspondingly, the target operation (on the "called" service provider) must have a dual "one-way" or "request-response" type signature to support the incoming invocation. However, it is not necessary that the types of the two signatures be the exact dual of each other, because the language allows for the mapping of input and output parameters. The only requirement is that the source is able to initiate the request, and the target is able to receive it. A more detailed discussion of plug links can be found in Section 3.5 "Recursive Composition Metamodel."

WSFL describes plug links using the `<plugLink>` element. The source and target of the link are specified via a nested `<source>` element and `<target>` element; each one of these identifies an operation in the interface of a service provider type using three attributes:

`operation`, `portType`, and either `serviceProvider` or `serviceProviderType`. A service provider type is specified when the endpoint operation belongs to an interface of the flow or global model that is being defined. Otherwise, the name of a service provider is specified.

Nested `<map>` elements can be used to adapt the input and output parameters of the source and target operations in case they don't match, see Section 4.5.3 "Data Links and Data Mapping." Finally, a locator can be provided to specify the actual endpoint address. For example, the following plug link uses a locator of type `mobility` that, based on the value of the `bankAccount` field of the source operation's output message of the plug link, determines the actual endpoint to interact with:

```
<plugLink>
   <source serviceProviderType="customer"
           portType="custSrv" operation="sendPAY"/>
   <target serviceProvider="supplier"
           portType="suppSrv" operation="rcvPAY"/>
   <map sourceMessage="paymentForm"
        targetMessage="paymentForm"/>
   <locator type="mobility"
            message="paymentForm"
            messagePart="recipient"
            dataField="bankAccount"/>
</plugLink>
```

The schema syntax for the `<plugLink>` element is given in the following code:

```
<complexType name="plugLinkType">
   <sequence>
      <element name="source" type="wsfl:endpointType"
               minOccurs="0"/>
      <element name="target" type="wsfl:endpointType"/>
      <element ref="map"
               minOccurs="0"/>
      <element name="locator" type="wsfl:locatorType"
               minOccurs="0"/>
   </sequence>
</complexType>
```

4.8 Global Model

A *global model* defines the interactions between partners in terms of client/server relationships between operations of their public interfaces; that is, the global model is a collection the service providers that interact and plug links that correlate (some of) the operations of their port types, indicating which operation is the originator of the interaction and which is the respondent. In other words, plug links are used to describe the association between "proxy" operations and the operations that are used to actually realize these proxies.

A plug link can carry a locator that is set by the source of the plug link and identifies the target of the plug link. Alternatively, a locator can be associated with the service provider; any type of locator can be used. A global model defines another service port type. In the following example, we define the service provider type `compoundBookOrder` that will provide the interface to buy books from two different booksellers. The public interface of this service provider type consists of the single operation `buy` of the only port type `lifeCycle`.

```
<serviceProviderType name="compoundBookOrder">
   <portType name="lifeCycle">
      <operation name="buy">
         <input message="budget"/>
         <output message="books"/>
      </operation>
   </portType>
</serviceProviderType>
```

The next example describes the interactions between partners that have to happen to order books. The global model `orderingSomeBooks` complies with the new service provider type `compoundBookOrder` defined above. The policy materialized by `orderingSomeBooks` is that dictionaries and poetry are always bought from different booksellers that have to provide the corresponding operations, thus we define two service providers called `bookseller01` and `bookseller02`. The two booksellers are statically bound, that is, the global model uses always the same booksellers (of course, any other locator could be used, making the example much more dynamic).

The service provider `bookLover` of service provider type `bookLoverPublic` exports the `spawn` operation of its lifecycle interface as `buy` operation of the public interface of the global model. Because no locator is defined for the book lover, a concrete instance will be determined at a later time. Finally, operations of the participating service providers are wired together by plug links. For plug links that target to `bookLover,` we use a locator of type `mobility` to enable a specification of the concrete recipient based on message content.

```
<globalModel name="orderingSomeBooks"
             serviceProviderType="compoundBookOrder">

   <serviceProvider name="bookseller01" type="bookseller">
      <locator type="static" service="muchToRead.com"/>
   </serviceProvider>

   <serviceProvider name="bookseller02" type="bookseller">
      <locator type="static" service="allYouCanRead.com"/>
   </serviceProvider>

   <serviceProvider name="bookLover" type="bookLoverPublic">
      <export>
         <source portType="lifeCycle"
                 operation="spawn"/>
         <target portType="lifeCycle"
                 operation="buy"/>
      </export>
   </serviceProvider>

   <plugLink>
      <source serviceProvider="bookLover"
              portType="bookRequester"
              operation="orderDictionary"/>
      <target serviceProvider="bookseller01"
              portType="processOrder"
              operation="receiveOrder"/>
   </plugLink>

   <plugLink>
      <source serviceProvider="bookLover"
              portType="bookRequester"
              operation="orderPoetry"/>
```

```
      <target serviceProvider=“bookseller02”
              portType=“processOrder”
              operation=“receiveOrder”/>
   </plugLink>

   <plugLink>
      <source serviceProvider=“bookseller01”
              portType=“processOrder”
              operation=“sendBooks”/>
      <target serviceProvider=“bookLover”
              portType=“bookRequester”
              operation=“receiveDictionary”/>
      <locator type="mobility"
               operation="receiveOrder"
               message="bookOrder"
               messagePart="whatever"
               dataField="customer"/>
   </plugLink>

   <plugLink>
      <source serviceProvider=“bookseller02”
              portType=“processOrder”
              operation=“sendBooks”/>
      <target serviceProvider=“bookLover”
              portType=“bookRequester”
              operation=“receivePoetry”/>
      <locator type="mobility"
               operation="receiveOrder"
               message="bookOrder"
               messagePart="whatever"
               dataField="customer"/>
   </plugLink>

</globalModel>
```

The above variant of the global model could be “bound” to a particular book lover as follows:

```
<globalModel name=“joeOdersSomeBooks” ref=“orderingSomeBooks”>
   <binding>
      <serviceProvider name=“bookLover”>
         <locator type=“static” service=“joe.com”/>
      </serviceProvider>
   </binding>
</globalModel>
```

The schema for the `<globalModel>` element is given in the next code example:

```
<complexType name="globalModelType">
   <choice>
      <sequence>
         <element name="serviceProvider"
                  type="wsfl:serviceProviderType"
                  maxOccurs="unbounded"/>
         <element name="plugLink"
                  type="wsfl:plugLinkType"
                  minOccurs=”0” maxOccurs="unbounded"/>
      </sequence>
      <element name=”binding”
               maxOccurs=”unbounded”>
         <complexType>
            <element name=”serviceProvider”
```

```
                    type=”serviceProviderRefType”
                    maxOcurs=”unbounded”/>
         </complexType>
      </element>
   </choice>
   <attribute name="name" type="NCName" use="required"/>
   <attribute name="serviceProviderType" type="Qname”/>
   <attribute name="ref" type="QName"/>
</complexType>

<complexType name=”serviceProviderRefType”>
   <complexContent>
      <restriction base=”serviceProviderType”/>
         <attribute name=”type” use=”prohibited”/>
      </restriction>
   </complexContent>
</complexType>
```

5 Appendix A: WSFL Schema

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsfl="http://schemas.xmlsoap.org/wsfl/"
        targetNamespace="http://schemas.xmlsoap.org/wsfl/"
        elementFormDefault="qualified">

<simpleType name="QNameList">
   <list itemType="QName"/>
</simpleType>

<simpleType name="NCNameList">
   <list itemType="NCName"/>
</simpleType>

<simpleType name="locatorTypeType">
   <restriction base="string">
      <enumeration value="static"/>
      <enumeration value="local"/>
      <enumeration value="any"/>
      <enumeration value="UDDI"/>
      <enumeration value="mobility"/>
   </restriction>
</simpleType>

<simpleType name="selectionPolicyType">
   <restriction base="string">
      <enumeration value="first"/>
      <enumeration value="random"/>
      <enumeration value="user-defined"/>
   </restriction>
</simpleType>

<simpleType name="bindTimeType">
   <restriction base="string">
      <enumeration value="startup"/>
      <enumeration value="firstHit"/>
   </restriction>
</simpleType>

<simpleType name="whenType">
   <restriction base="string">
      <enumeration value="deferred"/>
      <enumeration value="immediate"/>
   </restriction>
</simpleType>

<simpleType name="orderType">
   <restriction base="string">
      <enumeration value="LWW"/>
      <enumeration value="RFW"/>
      <enumeration value="random"/>
   </restriction>
</simpleType>

<element name="definitions" type="wsfl:definitionsType">
```

```
   <unique name="flowModelName">
      <selector xpath="flowModel"/>
      <field xpath="@name"/>
   </unique>
</element>

<complexType name="definitionsType">
   <sequence>
      <element name="import" type="wsfl:importType"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name=”serviceProviderType”
               type=”wsfl:serviceProviderTypeType"
               minOccurs=”0” maxOccurs="unbounded"/>
      <element ref="wsfl:flowModel"
               minOccurs=”0” maxOccurs="unbounded"/>
      <element ref="wsfl:globalModel"
               minOccurs=”0” maxOccurs="unbounded"/>
   </sequence>
   <attribute name="targetNamespace" type="uriReference"/>
</complexType>

<complexType name="importType">
   <attribute name="namespace" type="uriReference" use="required"/>
   <attribute name="location" type="uriReference" use="required"/>
</complexType>

<complexType name=”serviceProviderTypeType”>
   <sequence>
      <element name=”portType” type=”wsfl:portTypeType”
               minOccurs=”0” maxOccurs=”unbounded”/>
      <element name="import"
               minOccurs="0" maxOccurs=”unbounded”>
         <complexType>
            <attribute name=”portType” type=”QName”/>
         </complexType>
      </element>
   </sequence>
</complexType>

<complexType name="portTypeType">
   <complexContent>
      <extension base="wsdl:portTypeType">
         <sequence>
            <element name="import"
                     minOccurs="0" maxOccurs=”unbounded”>
              <complexType>
                 <attribute name="portType" type="QName"/>
                 <attribute name="operation" type="NCName"/>
              </complexType>
            </element>
         </sequence>
      </extension>
   </complexContent>
</complexType>

<element name="flowModel" type="wsfl:flowModelType">
   <key name="providerName">
      <selector xpath="serviceProvider"/>
      <field xpath="@name"/>
   </key>
   <key name="activityName">
```

```
      <selector xpath="activity"/>
      <field xpath="@name"/>
   </key>
   <unique name="controlLinkName">
      <selector xpath="controlLink"/>
      <field xpath="@name"/>
   </unique>
   <unique name="dataLinkName">
      <selector xpath="dataLink"/>
      <field xpath="@name"/>
   </unique>
   <keyref name="activityProviderRef" refer="providerName">
      <selector xpath="activity"/>
      <field xpath="@serviceProvider"/>
   </keyref>
   <keyref name="linkActivityRef" refer="activityName">
      <selector xpath="controlLink|dataLink"/>
      <field xpath="@source|@target"/>
   </keyref>
   <keyref name="implActivityRef" refer="providerName">
      <selector xpath="implement|import"/>
      <field xpath="@serviceProvider"/>
   </keyref>
</element>

<complexType name="flowModelType">
   <sequence>
      <element name="flowSource"
               type="wsfl:flowSourceType"
               minOccurs="0"/>
      <element name="flowSink"
               type="wsfl:flowSinkType"
               minOccurs="0"/>
      <element name="serviceProvider"
               type="wsfl:serviceProviderType"
               minOccurs=”1” maxOccurs="unbounded"/>
      <group ref="wsfl:activityFlowGroup"/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
   <attribute name="serviceProviderType" type="QName”/>
</complexType>

<group name="activityFlowGroup">
   <sequence>
      <element name="export" type="wsfl:exportType"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="activity" type="wsfl:activityType"
               minOccurs=”0” maxOccurs="unbounded"/>
      <element name="controlLink" type="wsfl:controlLinkType"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="dataLink" type="wsfl:dataLinkType"
               minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
</group>

<complexType name="serviceProviderType">
   <sequence>
      <element name="locator" type="wsfl:locatorType"
               minOccurs="0"/>
      <element name="export" type="wsfl:exportType"
               minOccurs="0" maxOccurs="unbounded"/>
```

```
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
   <attribute name="type" type="QName" use=”required”/>
</complexType>

<complexType name="locatorType">
   <sequence>
      <any namespace="##other"
           minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
   <attribute name="type" type="wsfl:locatorTypeType" />
   <attribute name="service" type="QName"/>
   <attribute name="bindTime" type="wsfl:bindTimeType"/>
   <attribute name="selectionPolicy"
              type="wsfl:selectionPolicyType"/>
   <attribute name="activity" type="NCName"/>
   <attribute name="message" type="NCName"/>
   <attribute name="messagePart" type="NCName" />
   <attribute name="dataField" type="string"/>
   <attribute name="default" type="QName"/>
   <attribute name="invoke" type="string"/>
</complexType>

<complexType name="flowSourceType">
   <sequence>
      <element ref=”wsdl:output”/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
</complexType>

<complexType name="flowSinkType">
   <sequence>
      <element ref=”wsdl:input”/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
</complexType>

<complexType name="endPointType">
   <attribute name="serviceProvider" type="NCName"/>
   <attribute name="serviceProviderType" type="NCName"/>
   <attribute name="portType" type="QName" use=”required”/>
   <attribute name="operation" type="NCName" use=”required”/>
</complexType>

<complexType name=”internalType”>
   <complexContent>
      <extension base="wsfl:endPointType"/>
         <sequence>
            <element name=”plugLink” type=”wsfl:plugLinkType”/>
         </sequence>
      </extension>
   </complexContent>
</complexType>>

<complexType name="joinType">
   <attribute name="condition" type="wsfl:NCNameList"
              use="required"/>
   <attribute name="when" type="wsfl:whenType"
              use="default" value="deferred"/>
</complexType>
```

```
<complexType name="materializeType">
   <choice>
      <element name="mapPolicy">
         <complexType>
            <attribute name="order" type="wsfl:orderType"
                       use="default" value="LWW"/>
         </complexType>
      </element>
      <element name="construction">
         <complexType>
            <attribute name="type" type="string"
                       use="default" value="XSLT"/>
            <attribute name="location" type="string"/>
         </complexType>
      </element>
   </choice>
</complexType>

<complexType name="activityType">
   <complexContent>
      <extension base="wsdl:operationType">
         <sequence>
            <element name=”performedBy”>
               <complexType>
                  <attribute name=”serviceProvider”
                             type=”NCName”/>
               </complexType>
            </element>
            <element name="implement">
               <complexType>
                  <choice>
                     <element name="internal"
                              type=”wsfl:internalType”>
                     <element name="export"
                              type="wsfl:exportType"/>
                  </choice>
               </complexType>
            </element>
            <element name="join" type="wsfl:joinType"
                     minOccurs="0"/>
            <element name="materialize"
                     type="wsfl:materializeType"
                     minOccurs="0"/>
            <any namespace="##other"
                 minOccurs="0" maxOccurs="unbounded"/>
         </sequence>
         <attribute name="name" type="NCName"/>
         <attribute name="exitCondition" type="string"/>
      </extension>
   </complexContent>
</complexType>

<complexType name="linkType">
   <attribute name="name" type="NCName"/>
   <attribute name="source" type="NCName"/>
   <attribute name="target" type="NCName"/>
</complexType>

<complexType name="controlLinkType">
   <complexContent>
      <extension base="linkType">
```

```
         <attribute name="transitionCondition" type="string"/>
         <attribute name="result" type="NCName"/>
      </extension>
   </complexContent>
</complexType>

<complexType name="dataLinkType">
   <complexContent>
      <extension base="linkType">
         <sequence>
            <element ref="map" minOccurs="0"/>
         </sequence>
      </extension>
   </complexContent>
</complexType>

<complexType name="plugLinkType">
   <sequence>
      <element name="source" type="wsfl:endpointType"
               minOccurs="0"/>
      <element name="target" type="wsfl:endpointType"/>
      <element ref="map" minOccurs="0"/>
      <element name="locator" type="wsfl:locatorType"
               minOccurs="0"/>
   </sequence>
</complexType>

<element name="map">
   <complexType>
      <attribute name="sourceMessage" type="NCName"/>
      <attribute name="targetMessage" type="NCName"/>
      <attribute name="sourcePart" type="NCName"/>
      <attribute name="targetPart" type="NCName"/>
      <attribute name="sourceField" type="NCName"/>
      <attribute name="targetField" type="NCName"/>
      <attribute name="converter" type="string"/>
   </complexType>
</element>

<complexType name="exportType">
   <sequence>
      <element name="source" type="endPointType"
               minOccurs="0"/>
      <element name="target" type="endPointType"/>
      <element ref="wsfl:map"
               minOccurs="0" maxOccurs="unbounded"/>
      <element name="plugLink" type="wsfl:plugLinkType"
               minOccurs="0"/>
   </sequence>
   <attribute name="lifecycleAction" type="NCName"/>
</complexType>

<element name="globalModel" type="wsfl:globalModelType">
   <unique name="globalProviderName">
      <selector xpath="serviceProvider"/>
      <field xpath="@name"/>
   </unique>
</element>

<complexType name="globalModelType">
   <choice>
```

```
      <sequence>
         <element name="serviceProvider"
                  type="wsfl:serviceProviderType"
                  maxOccurs="unbounded"/>
         <element name="plugLink" type="wsfl:plugLinkType"
                  minOccurs=”0” maxOccurs="unbounded"/>
      </sequence>
      <element name=”binding” maxOccurs=”unbounded”>
         <complexType>
            <element name=”serviceProvider”
                     type=”serviceProviderRefType”
                     maxOcurs=”unbounded”/>
         </complexType>
      </element>
   </choice>
   <attribute name="name" type="NCName" use="required"/>
   <attribute name="serviceProviderType" type="Qname”/>
   <attribute name="ref" type="QName"/>
</complexType>

<complexType name=”serviceProviderRefType”>
   <complexContent>
      <restriction base=”serviceProviderType”/>
         <attribute name=”type” use=”prohibited”/>
      </restriction>
   </complexContent>
</complexType>

<simpleType name="state" base="string">
   <enumeration value="Running"/>
   <enumeration value="Suspended"/>
</simpleType>

<simpleType name="Success" base="string">
   <enumeration value="OK"/>
</simpleType>

<element name="FlowInstanceID" type="string"/>
<element name="FlowInstanceState" type="wsfl:state"/>
<element name="FlowInstanceLastModificationTime" type="dateTime"/>
<element name="FlowInstanceCreationTime" type="dateTime"/>

<complexType name="SpawnResult">
   <sequence>
      <element ref="wsfl:FlowInstanceID"/>
      <element ref="wsfl:FlowInstanceCreationTime"
               minOccurs="0" maxOccurs="1"/>
   </sequence>
</complexType>

<complexType name="EnquiryInput">
   <sequence>
      <element ref="wsfl:FlowInstanceID"/>
   </sequence>
</complexType>

<complexType name="EnquiryResult">
   <sequence>
      <element ref="wsfl:FlowInstanceID"/>
      <element ref="wsfl:FlowInstanceState"/>
      <element ref="wsfl:FlowInstanceCreationTime"
```

```
                minOccurs="0" maxOccurs="1"/>
      <element ref="wsfl:FlowInstanceLastModificationTime"
                minOccurs="0" maxOccurs="1"/>
   </sequence>
</complexType>

<complexType name="Fault">
   <sequence>
      <element name="MainCode" type="integer"/>
      <element name="SubCode" type="integer"
                minOccurs="0" maxOccurs="1"/>
      <element name="MessageText" type="string”
                minOccurs="0" maxOccurs="1"/>
   </sequence>
</complexType>

</schema>
```

6 Appendix B: Internal Activity Implementations

Activity implementations are specified as operations of port types. When an activity implementation is an internal application, the referenced operation and port type does not belong to the public interface of the flow model. This is specified by the `<internal>` element, that is, the corresponding operation is not exported. The service provider type that is referenced by the corresponding attribute of the `<internal>` element typically collects port types that define the proxies needed for accessing the internal applications. These proxies are typically plug-linked in an “inline” manner to operations of port types that belong to “internal” service providers defining the interfaces of the stubs of the internal applications.

```
<flowModel name=“bookLover” serviceProviderType=“bookLoverPublic”>

   <activity name=“selectBook”>
      <input message=“comma123”/>
      <output message=“comma123”/>
      <implement>
         <internal serviceProviderType=“bookLoverPrivate”
                   portType=“bookStuff” operation=“selectBook”>
            <plugLink>
               <source serviceProviderType=“bookLoverPrivate”
                       portType=“bookStuff” operation=“selectBook”/>
               <target serviceProviderType =“someCicsTransactions”
                       portType=“cicsy” operation=“T123”/>
            </plugLink>
         <internal>
      </implement>
   </activity>

</flowModel>
```

The corresponding WSDL definitions include extensibility elements that provide all the required information to properly access the internal applications. For example, information about what the command line looks like in order to invoke an EXE file, which CICS subsystem hosts the transaction to be invoked, how the corresponding COMMAREA must be laid out, and so on. The following sections introduce the required extensibility elements for some executables that are frequently found for internal applications.

6.1 EXE Files

The following XML document illustrates the definition of EXE/CMD files as operations in WSDL. It defines a service that provides for the creation, editing, and deletion of documents; defined as operations through the appropriate operation specification within the `PortType` section. An appropriate program is called for document creation and modification. Deletion is handled through an appropriate command file (that for example, may invoke the appropriate `Delete` function of the operating system).

The appropriate binding extensions for editing a document specifies the EXE file to be invoked, identified through `pathAndFileName,` which specifies the location where the executable resides as well as the name of the executable (in our example, `word`). The appropriate input specification extensions indicate through the encoding specification that the input message should be passed as a command string.

```
<definitions name=”DocumentProcessing”
   targetNamespace=”http://example.com/documentProcessing.wsdl”
   xmlns:exe=”http//schemas.xmlsoap.org/wsdl/exe”/>
```

```
   <types>
      <element name=”Document”>
         <complexType>
            <all>
                <element name=”DocumentName” type=”string”/>
            </all>
         </complexType>
      </element>
   </types>

   <message name=”DocumentIdentification”>
      <part name=”body” element=”Document”/>
   </message>

   <portType name=”DocumentProcessingPortType”
      <operation name=”CreateDocument”/>
      <operation name=”EditDocument”>
          <input message=”tns:DocumentIdentification”/>
      </operation>
      <operation name=”DeleteDocument”>
          <input message=”tns:DocumentIdentification”/>
      </operation>
   </portType>

   <binding name=”DocumentProcessingBinding”
            type=”DocumentProcessingPortType”>
      <operation name=”CreateDocument”>
          <exe:operation pathAndFileName=”word.exe”
                         startInForeground=”yes”
                         style=”visible”
                         executionMode=”normal”/>
      </operation>
      <operation name=”EditDocument”>
          <exe:operation pathAndFileName=”word.exe”
                         startInForeground=”yes”
                         inheritEnvironment=”yes”
                         style=”visible”
                         executionMode=”normal”/>
          <input>
              <exe:input encoding=”commandParameters”/>
          </input>
      </operation>
      <operation name=”DeleteDocument”>
          <cmd:operation pathAndFileName=”deldoc.cmd”/>
      </operation>
   </binding>

   <service name=”DocumentProcessingService”>
      <documentation>Document Processing Service</documentation>
      <port name=”DocumentProcessingPort”
            binding=”tns:DocumentProcessingBinding/>
   </service>

</definitions>
```

6.2 Customer Information Control System (CICS) Programs

The following example illustrates the usage of two CICS transactions to manage bank accounts. Two operations are defined, a `WithdrawMoney` operation to withdraw money from a specified account and `DepositMoney` to deposit money into a specified account. As specified in the bindings, the `WithdrawMoney` operation is implemented through a CICS transaction called `WMON` and the `DepositMoney` operation through a CICS transaction called `DMON`.

The encoding specification for the input messages indicates that the input message should be passed in the layout of a CICS COMMAREA; certain default assumptions are used, for example, that the sequence of the fields in the COMMAREA is the one specified for the message. The CICS system that hosts those transactions is identified as in the port definition of the service through an appropriate CICS extension that identifies the appropriate `APPLID`. This specification allows clients to locate the appropriate CICS subsystem and invoke the transactions.

```
<definitions name="AccountProcessing"
   targetNamespace="http://example.com/accountProcessing.wsdl"
   xmlns:exe="http://schemas.xmlsoap.org/wsdl/CICS"/>

   <types>
      <element name="Account">
         <complexType>
            <all>
                <element name="AccountId" type="string"/>
            </all>
         </complexType>
      </element>
      <element name="Amount">
         <complextType>
            <all>
                <element name="Value" type="positiveInteger"/>
                <element name="Currency" type="string"/>
            </all>
         </complexType>
      </element>
   </types>

   <message name="AccountOperation">
      <part name="part1" element="Account">
      <part name="part2" element="Amount"
   </message>

   <portType name="AccountPortType"
      <operation name="DepositMoney">
          <input message="tns:AccountOperation"/>
      </operation>
      <operation name="WithdrawMoney"
          <input message="tns:AccountOperation"/>
      </operation>
   </portType>

   <binding name="AccountBinding" type="AccountPortType">
      <operation name="DepositMoney">
          <CICS:operation transactionID="DMON"/>
          <input>
              <CICS:input encoding="COMMAREA"/>
```

```
          </input>
      </operation>
      <operation name=”WithdrawMoney”>
          <CICS:operation transactionID=”WMON”/>
          <input>
              <CICS:input encoding=”COMMAREA”/>
          </input>
      </operation>
   </binding>

   <service name=”AccountService”>
      <documentation>Account Service via CICS</documentation>
      <port name=”AccountPortType”
            binding=”tns:AccountBinding>
          <CICS:port APPLID=”IPWAYCIA”/>
      </port>
   </service>

</definitions>
```

6.3 Java Classes

This example illustrates a Java binding for a credit card verification service. It offers just one operation, which takes credit card information as input and produces a status message as output. This service is bound to a concrete implementation provided by a local Java class. WSDL types (except for built-in schema types) that are part of the service are mapped to local Java classes. Operations within the port types are mapped to methods in the Java class that provides the service. Messages intended for this service are de-serialized into Java types and the appropriate method of the provider class is invoked with the parameters. Conversely, the output resulting from the invocation of the Java method is serialized to an XML message of the appropriate type.

```
<definitions name="CreditCardVerifier"
   targetNamespace="http://example.com/creditCardVerification.wsdl"
   xmlns:java="http://schemas.xmlsoap.org/wsdl/java"/>

   <types>
      <schema>
         <complexType name="creditCard">
            <element name="number" type="integer"/>
            <element name="name" type="string"/>
            <element name="expirationDate" type="gYearMonth"/>
         </complexType>
      </schema>
      <schema>
         <simpleType name="cardStatus">
            <restriction base="string">
               <enumeration value="STATUS_OK"/>
               <enumeration value="STATUS_INVALID_NUMBER"/>
               <enumeration value="STATUS_INVALID_NAME"/>
               <enumeration value="STATUS_EXPIRED"/>
            </restriction>
         </simpleType>
      </schema>
   </types>

   <message name="CardInformation">
      <part name="cardinfo" type="tns:creditCard"/>
   </message>
```

```
   <message name="CardStatus">
      <part name="status" type="tns:cardStatus"/>
   </message>

   <port type name="CreditCardVerificationPortType">
      <operation name="verifyCard">
         <input message="CardInformation"/>
         <output message="CardStatus"/>
      </operation>
   </port type>

   <binding name="LocalCardVerifier"
            type="tns:CreditCardVerificationPortType">
      <operation name="verifyCard">
         <java:operation  javamethod="verifyCreditCard"/>
         <input>
            <java:typemapping name="tns:creditCard"
                  class="com.example.verifier.CreditCard"
                  serializer="com.example.verifier.CardSerializer"

deserializer="com.example.verifier.CardSerializer"/>
         </input>
         <output>
            <java:typemapping name="tns:cardStatus"
                 class="com.example.verifier.Status"
                 serializer="com.example.verifier.StatusSerializer"

deserializer="com.example.verifier.StatusSerializer"/>
         </output>
      </operation>
   </binding>

   <service name="CreditCardVerificationService">
      <port name="CreditCardVerificationPort"
            binding="LocalCardVerifier">
         <java:provider class="com.example.verifier.CardVerifier"/>
      </port>
   </service>

</definitions>
```

7 Appendix C: Endpoint Property Extensibility Elements

Activities typically represent business tasks or business interactions. As such, activities are associated with information that represents their business context. As discussed in Section 3.1.1.16 “Endpoint Properties,” the business context has many aspects. In this section, we very briefly sketch only a few sample endpoint properties that are aspects of such a business context. We do not even try to be exhaustive here because we assume that an appropriate “Web Services Endpoint Language” (WSEL) will define in future such a business context. Thus, we introduce the endpoint properties as extensibility elements in WSFL.

7.1 Execution Limits

An activity can be associated with a duration controlling the maximum time after which a result is expected. This duration is specified by the `<duration>` element. Or the activity is associated with a `<retry>` element that specifies the maximum number of attempts that are made to invoke the endpoint of the plug link of the activity’s implementation.

7.2 Escalation

When this threshold is exceeded, a person from the organization running the invoking flow should be notified. This person is specified through the `<escalate>` element. The person determined would typically contact a member from the service provider organization to check what is going on. This contact can be derived from properties of the port type that is plug-linked with the operation that is implementing the activity.

7.3 Observation

A person from the organization running the invoking flow may track the execution of an activity (for example, to detect out-of-line situations in advance). This person is specified through the `<observed>` element. The observing person will act as the contact point for a representative of the service provider organization, or will monitor the state of the activity, for example.

7.4 Contacts

Contacts are relevant information defined for a service provider as well as for service requestors. This information is typically specified through a nested `<staff>` element that refers to a person or declarative description of a person.

Many more service-level parameters can be specified, for example, for contractual agreements, costs, and so on. All of this information is not specified in WSFL, but using extension elements provided by the envisioned WSEL.

The following sample encoding uses the extensibility elements introduced above:

```
<activity name="processPO”>
   <-- extensibility elements – Details defined in WSEL      -->
   <wsel:duration limit=”30” metric=”minutes”/>
```

```
   <wsel:retry maxNumber=”10”/>

   <wsel:escalate>
      <wsel:staff
            who=”select PID from Person where skill > 15”
            Invoke=”c:\programs\org_query.exe”/>
   </wsel:escalate>

   <wsel:observed>
      <wsel:staff
            who=” select PID
                  from Flows
                  where FlowName= “TotalSupplyFlow” ”
            invoke=”c:\programs\org_query.exe”/>
   </wsel:observed>

</activity>
```

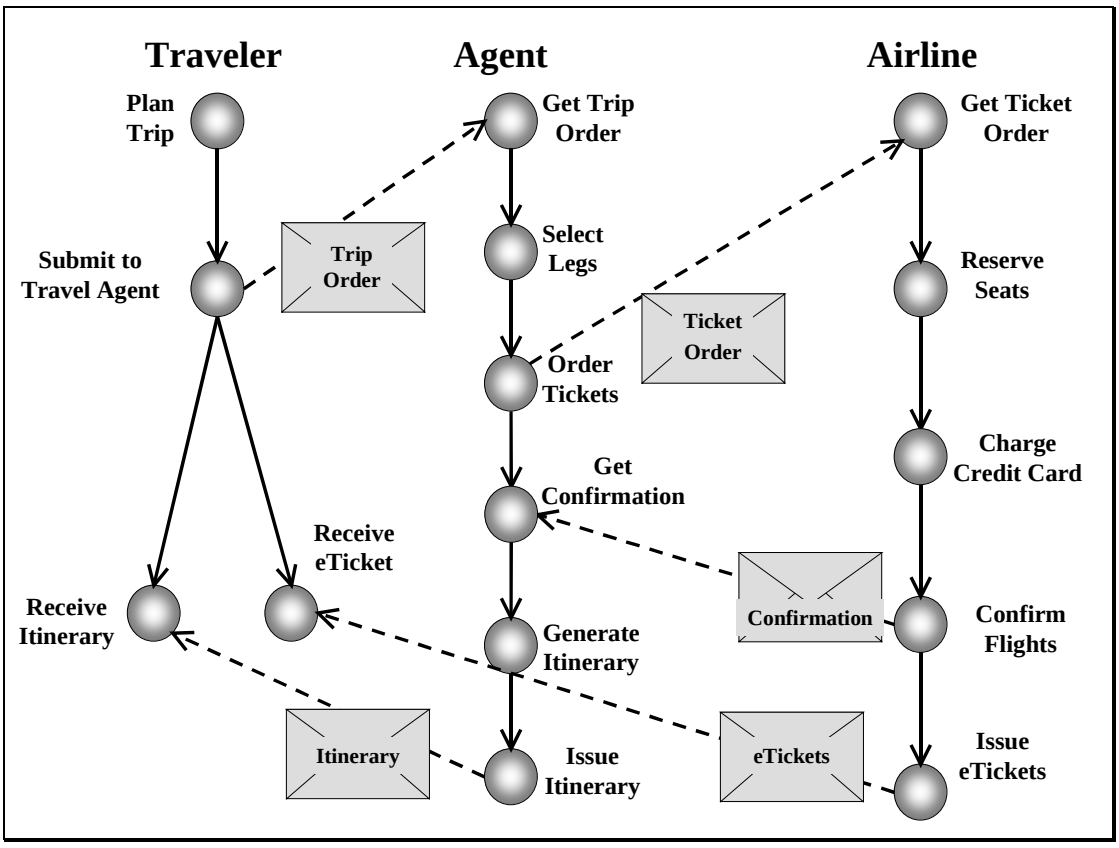
8 Appendix D: The Ticket-Order Example

We are now providing major aspects of a more complex example of a business process between multiple service providers in WSFL. The example is for ordering tickets over the Internet. First, we will sketch the overall process briefly. Next, we explain more details by describing the messages and port types used, the flow models describing what takes place at two of the participating service provider, and a global model showing the binding between two of the three partners.

8.1 Overview

The overall flow of this example is as follows: A traveler will plan her trip by specifying the various stages of his overall journey and all of its participants. For each of the stages, the traveler specifies the location that she wants to visit as well as the date when she wants to begin and the date when she wants to end the particular stay. When the travler is finished with this, she sends this information together with the list of all participants of the trip as well as the information about the credit card to be charged for the ordered tickets to the travel agent. The credit card information has been passed when the business process was instantiated. Next, the traveler will await the submission of the electronic tickets as well as the final itinerary for the trip.

When the agent receives the traveler's trip order, he will determine the legs for each of the stages, which includes an initial seat reservation for each of the participants as well as the rate chosen. To actually make the corresponding ticket orders the agent submits these legs together with the information about the credit card to be charged to the airline company. Then, the agent will wait for the confirmation of the flights, which especially includes the actual seats reserved for each of the participants. This information is completed into an itinerary, which is then sent to the traveler.



When the airline receives the ticket order submitted by the agent, the requested seats will be checked and if available assigned to the corresponding participants. After that, the credit card will be charged, and the updated leg information is sent back to the agent as confirmation of the flights. After that, the airline sends the electronic tickets by e-mail to the traveler. Information about the recipient of the tickets has been specified by the traveler when instantiating the trip order process and this information is passed to the agent as well as to the airline.

We approach this example in several stages:

1. Define the messages used
2. Define the port types in WSDL
3. Define the “local” airline and agent business processes as WSFL flow models
4. Define the composition of the airline and agent flow as a WSFL global model

In this section, we assume that all information provided in the graphical picture of the flow above is public information to be expressed in WSFL and supporting WSDL and WSEL, and shared between participants in the flow. There may be additional implementation information inside the flow models, which is private to the owner of the flow, but it is beyond this example as described in this appendix. In order to improve the readability the of the WSFL example, we will make the assumption that all the messages, port types, service provider types, and flow models are defined within the same target namespace, which will be assigned the namespace prefix `tio`. Also for simplicity, we will assume that no default namespace is defined.

8.2 Messages for the Ticket-Order Example

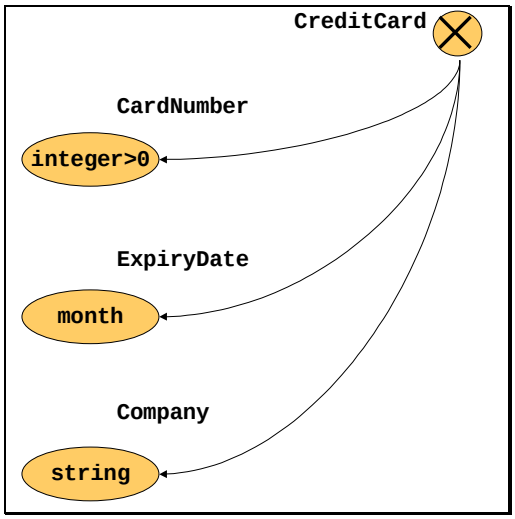
This section lists the messages for the ticket-order example.

8.2.1 Short Description and Graphical Definition

We introduce each of the basic message elements used to interact between the flows with a text description and a diagram. Formal definitions are provided as WSDL text in the following subsection.

8.2.1.1 The Credit Card Message

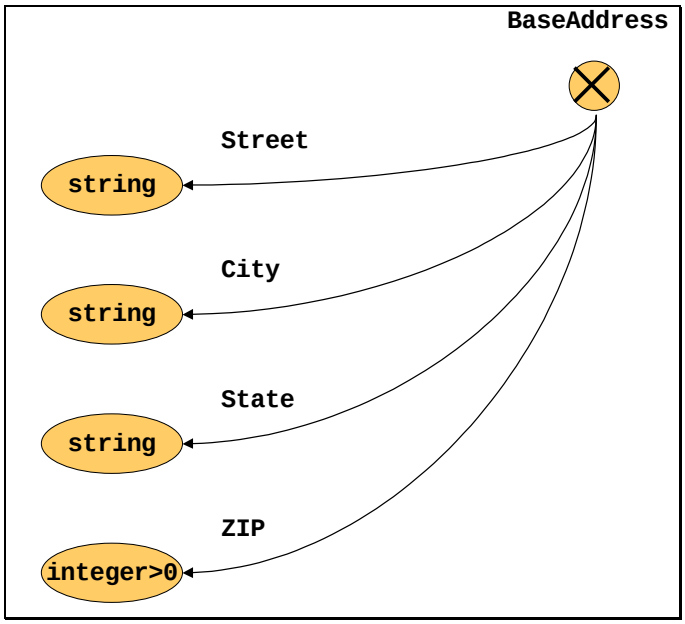
When the traveler instantiates a trip-order flow, she will be immediately asked to provide the information about the credit card to be charged. In fact, this data is part of the signature of the corresponding lifecycle operation for kicking off the flow.



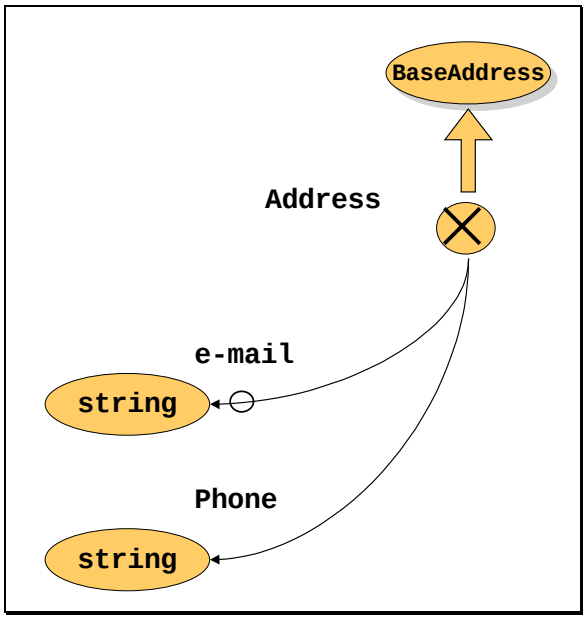
8.2.1.2 The Participants Message

The people who will go on the trip are referred to as the *participants*. We are now going to define the details of the information about participants.

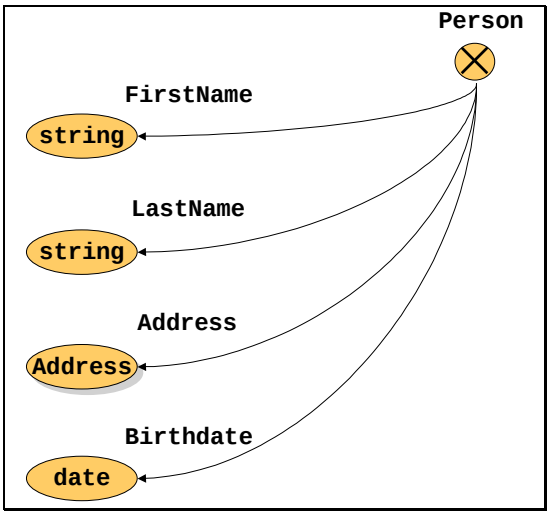
We first define the structure of a base address, which simply aggregates street, city, state, and ZIP information. Obviously, our example is geared towards the U.S.; in order to be also applicable in other countries, different kinds of addresses have to be defined (for example, through appropriate `CHOICE` or `EXTENSION` mechanisms).



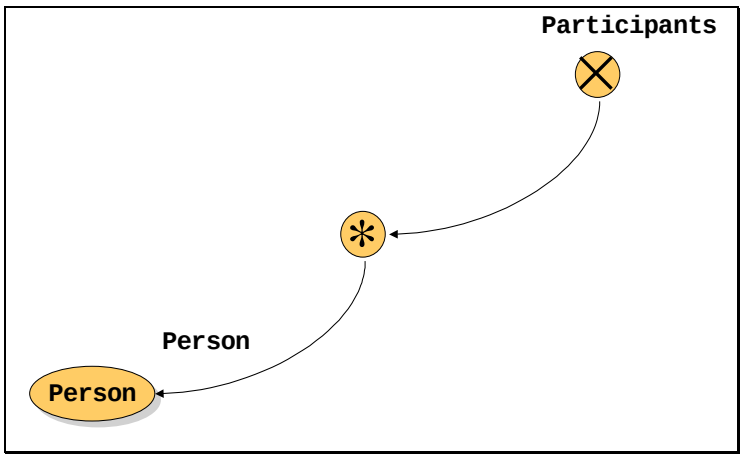
The base address is extended into an address, which includes an optional e-mail address as well as a mandatory phone number.



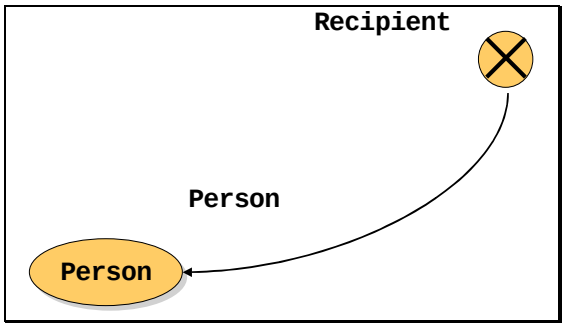
This address is then used to define a person: A person has a first name and a last name, an address as well as a birth date.



Then we are ready to define the `Participants` of the trip as a set of one `Person` or more.

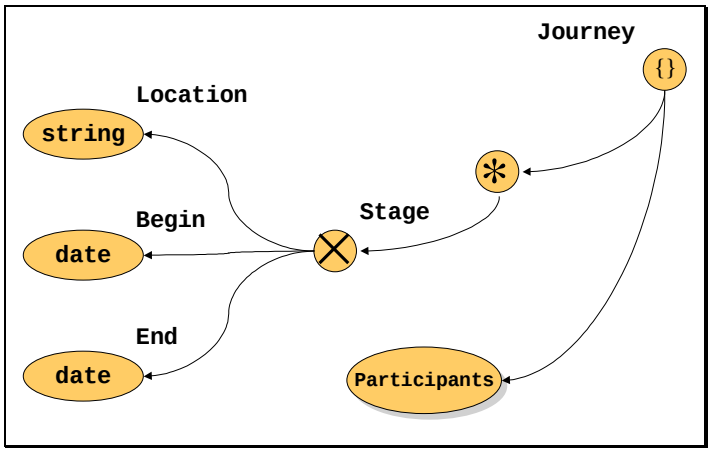


Finally, when the traveler instantiates a trip-order flow, she will be immediately asked to provide the information about the person to whom the final itinerary as well as the tickets have to be sent, i.e. the `Recipient`. This data is part of the signature of the corresponding lifecycle operation for kicking off the flow.



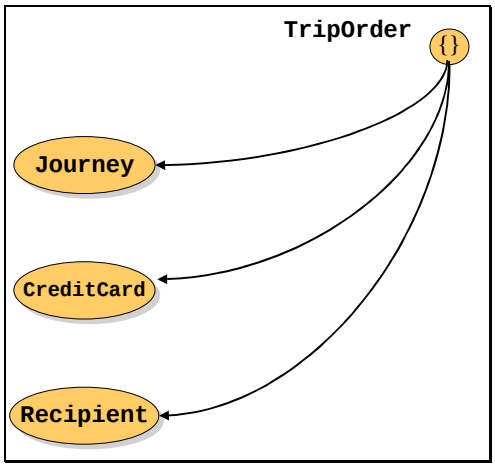
8.2.1.3 The Journey Message

A `Journey` consists of a set of one or more stages and the participants (see Section 8.2.1.2 “The Participants Message”) of the journey. A `Stage` is a tuple consisting of a location and the begin date and end date of the stay at that particular location.



8.2.1.4 The Trip Order Message

A `TripOrder` message is sent from the ordering traveler to the agent. It takes the information about a journey and adds the information about the credit card that is to be used to pay for the costs of the corresponding flight tickets of all participants. The purpose of the `Recipient` element is mobility: The included `e-mail` element will be used as the address to which the electronic tickets have to be delivered.

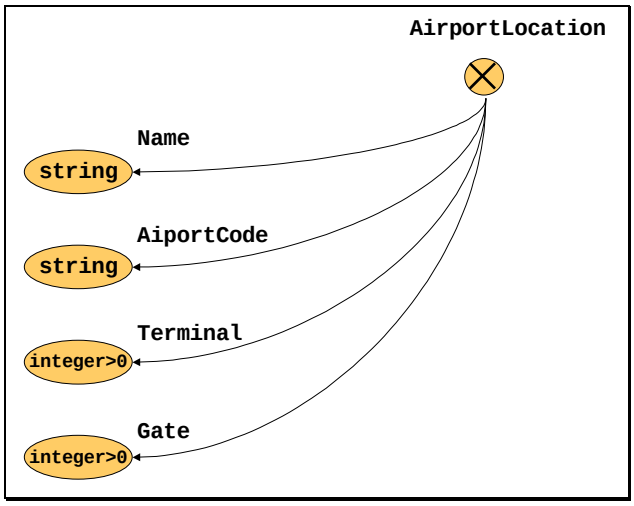


8.2.1.5 The Legs Message

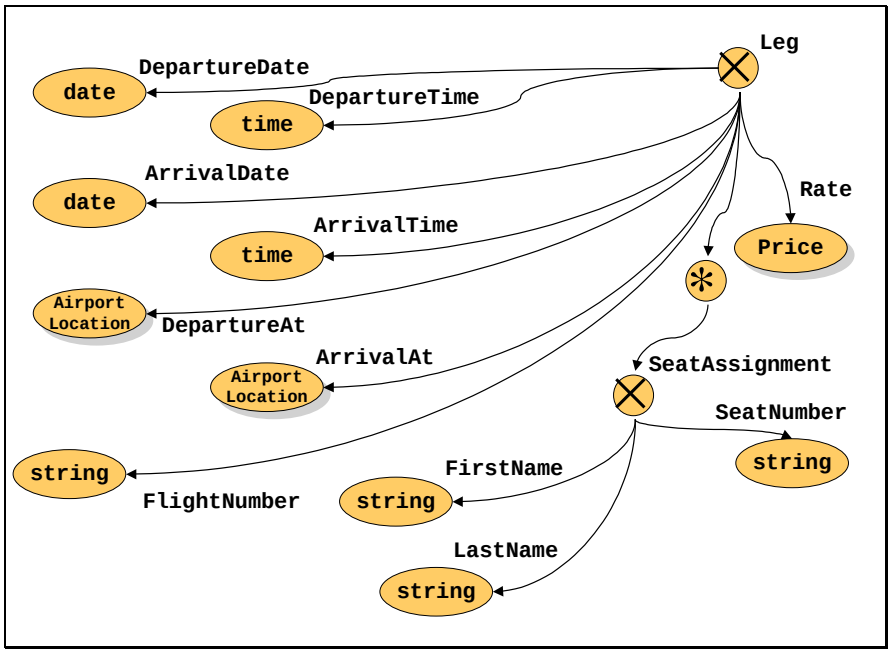
A leg describes the details of a particular flight connection like airport information as well as seat assignment for each of the participants. These details follow next.

We will define `Price` as a decimal of scale two.

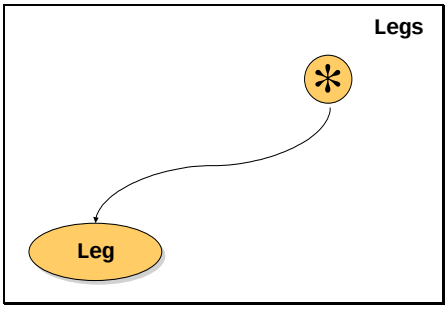
The airport location provides the name of an airport, its internationally unique airport code, and the terminal and gate information a flight departs or arrives on.



A leg, that is, a particular flight connection of the trip, shows the departure date and time as well as arrival date and time for this flight connection. Furthermore, the corresponding airport location information is given for where the flight arrives and where it departs. A string consisting of two characters and at least one to four digits represents the flight number of this flight. The `SeatAssignment` element collects for each participant the first and last name, the seat number assigned and the rate to be paid for the flight. The seat number is a string consisting of two or three digits (prepared already for the new generation of large airplanes) followed by one character.

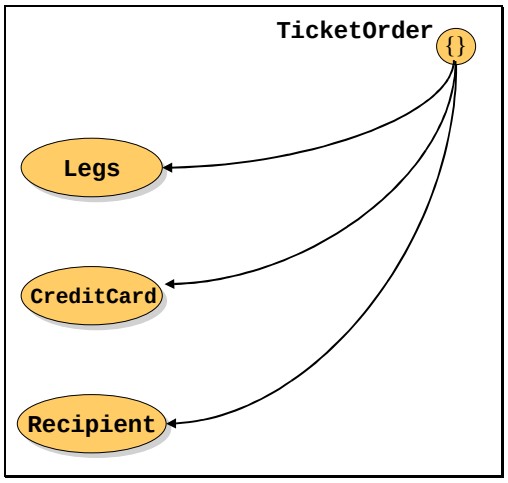


The `Legs` message finally collects all the legs of the trip.



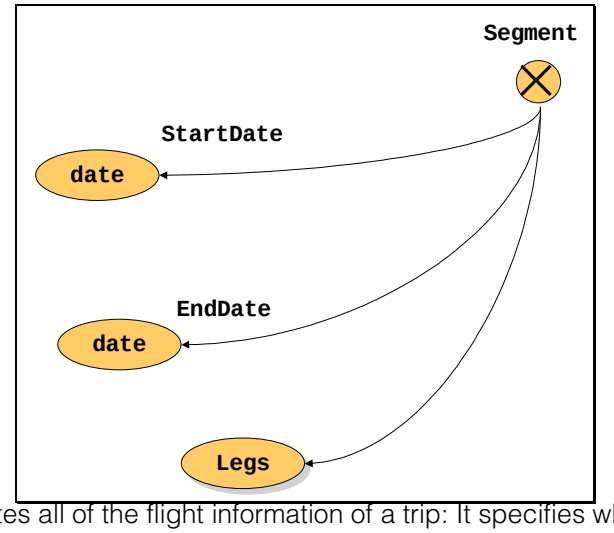
8.2.1.6 The Ticket Order Message

A `TicketOrder` message is sent from the agent to the airline company. It contains information about the credit card to be charged by the airline as well as the legs that the agent already determined to be appropriate to cover the trip. The `Recipient` element is for mobility: The included `e-mail` element will be used by the airline as the address to which the electronic tickets have to be delivered.

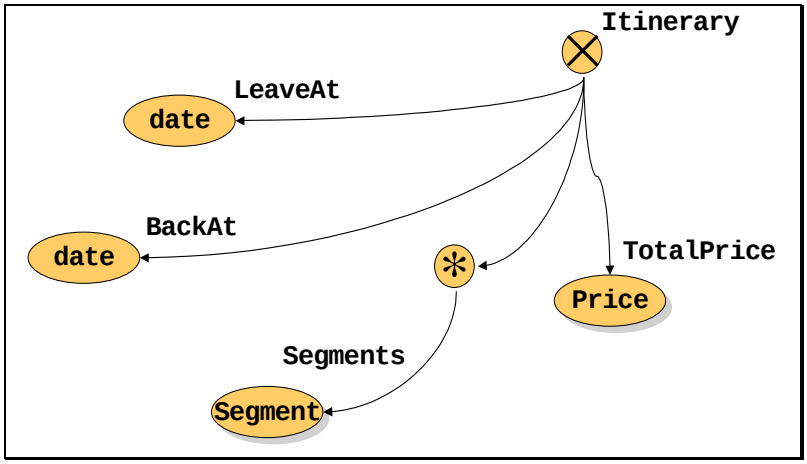


8.2.1.7 The Itinerary Message

A segment collects all the flight information that corresponds to a particular location that the traveler specified when making a trip request: The date on which to start at the current location, the date on which to finally arrive at the target location, and the legs required to get there.



An itinerary aggregates all of the flight information of a trip: It specifies when the overall trip will begin and when it will end, and it provides all segments of the trip and its total price.



8.2.2 Additional Messages

Additional messages need to be defined for the input and output data signatures of each of the activities defined in the airline and ticket agent flows.

8.2.3 Message Definition File

The messages used in our example are defined in a file named TicketOrder.xml. For these messages, a separate namespace has been created.

```
<?xml version="1.0"?>
<definitions name=“totalTravel“
 targetNamespace=
 "http://www.TravelLuck.com/WebServices/Messages/TotalTravel"
 xmlns:tio=
 “http://www.TravelLuck.com/WebServices/Messages/TotalTravel”
 xmlns=“”>
```

```
<types>
<schema
     xmlns= “http://www.w3.org/2000/10/XMLSchema”
     targetNamespace =
       “http://www.TravelLuck.com/WebServices/Messages/TotalTravel”
     xmlns:tio=
       “http://www.TravelLuck.com/WebServices/Messages/Totaltravel”>

<element name="CreditCard">
    <complexType>
        <sequence>
            <element name="CardNumber" type="nonNegativeInteger"/>
            <element name="ExpiryDate" type="month"/>
            <element name="Company" type="string"/>
        </sequence>
    </complexType>
</element>

<complexType name="BaseAddress">
    <sequence>
        <element name="Street" type="string"/>
        <element name="City" type="string"/>
        <element name="State" type="string"/>
        <element name="ZIP" type="nonNegativeInteger"/>
    </sequence>
</complexType>

<complexType name="Address">
    <complexContent>
        <extension base="tio:BaseAddress">
            <sequence>
                <element name="e-mail" type="string" minOccurs="0"/>
                <element name="Phone" type="string"/>
            </sequence>
        </extension>
    </complexContent>
</complexType>

<complexType name=”Person”>
   <sequence>
      <element name="FirstName" type="string"/>
      <element name="LastName" type="string"/>
      <element name="Address" type="tio:Address"/>
      <element name="BirthDate" type="date"/>
   </sequence>
</complexType>

<element name="Participants" type="tio:Person"
         minOccurs="1" maxOccurs="unbounded"/>

<element name="Recipient" type="tio:Person"/>

<element name="Journey">
    <complexType>
        <all>
           <element name="Stage"
                    minOccurs="1" maxOccurs="unbounded">
               <complexType>
                   <sequence>
                       <element name="Location" type="string"/>
                       <element name="Begin" type="date"/>
```

```
                        <element name="End" type="date"/>
                    </sequence>
                </complexType>
            </element>
            <element ref="tio:participants"/>
        </all>
    </complexType>
</element>

 <!--============================================================ -->
 <!-- tripOrder is the basic “where to” information supplied by  -->
 <!--            the traveller to the travel agent              -->
 <!--============================================================ -->

<element name="tripOrder">
    <complexType>
        <all>
            <element ref="tio:Journey"/>
            <element ref="tio:CreditCard"/>
            <element ref=”tio:Recipient”/>
        </all>
    </complexType>
</element>

<complexType name="AirportLocation">
    <sequence>
        <element name="Name" type="string"/>
        <element name="AiportCode" type="string"/>
        <element name="Terminal" type="positiveInteger"/>
        <element name="Gate" type="positiveInteger"/>
    </sequence>
</complexType>

<complexType name="Leg">
    <sequence>
        <element name="DepartureDate" type="date"/>
        <element name="DepartureTime" type="time"/>
        <element name="ArrivalDate" type="date"/>
        <element name="ArrivalTime" type="time"/>
        <element name="DepartureAt" type="tio:AirportLocation"/>
        <element name="ArrivalAt" type="tio:AirportLocation"/>
        <element name="FlightNumber">
            <simpleType>
                <restriction base="string">
                    <pattern value="\S{2}\d{1,4}"/>
                </restriction>
            </simpleType>
        </element>
    </sequence>
</complexType>

<element name="Legs" type="tio:Leg"
         minOccurs="1" maxOccurs="unbounded"/>

<element name= “seatAssignment” >
   <complexType>
      <sequence>
         <element name="FirstName" type="string"/>
         <element name="LastName" type="string"/>
         <element name="SeatNumber">
            <simpleType>
```

```
               <restriction base="string">
                  <pattern value="\d{2,3}\S{1}"/>
               </restriction>
            </simpleType>
         </element>
         <element name="Rate" type="string"/>
      </sequence>
   </complexType>
</element>

<element name= “seatAssignments” type=”tio:seatAssignment”
         minOccurs=”1” maxOccurs=”unbounded”/>

 <!--============================================================ -->
 <!-- ticketOrder (including legs) flows from agent to airline  -->
 <!--============================================================ -->

<element name="ticketOrder">
    <complexType>
        <all>
            <element ref="tio:CreditCard"/>
            <element ref="tio:Legs"/>
            <element ref=”tio:Recipient”/>
        </all>
    </complexType>
</element>

<simpleType name="Price">
    <restriction base="decimal">
        <scale value="2"/>
    </restriction>
</simpleType>

<simpleType name= “eTicket” type=”string”/>

<simpleType name=“chargeTxId” type=”integer”/>

<element name=”ticketOrderRecord”>
  <complexType>
    <all>
      <element name= “request” type= “tio:tripOrder”/>
      <element name= “agentWorkId” type= “wsfl:FlowInstanceId”/>
      <element name= “ourTicketOrder” type= “tio:ticketOrder”/>
      <element name= “airlineWorkId” type=“wsfl:FlowInstanceId”/>
    </all>
  </complexType>
</element>

<!--============================================================ -->
<!--Confirmation (with seat assignments, prices and record of     -->
<!--charge to customer ) flows from airline to agent              -->
<!--============================================================ -->

<element name= “confirmation”>
 <complexType>
    <all>
       <element name= “Request” type= “tio:ticketOrder”/>
       <element name= “airlineWorkId” type= “wsfl:FlowInstanceId”/>
       <element name= “seating” type= “tio:seatAssignments”/>
       <element name= “chargeTxId” type=”integer”/>
    </all>
```

```
  </complexType>
</element>

<!--============================================================ -->
<!--Itinerary (with original order, ticket information, confirmed -->
<!--seat assignments, total price and record of charge to        -->
<!--customer ) flows from airline to agent                       -->
<!--============================================================ -->

<element name= “itinerary”>
 <complexType>
  <all>
    <element name= “trip” type= “tio:tripOrder”/>
    <element name= “agentWorkId” type= “wsfl:FlowInstanceId”/>
    <element name= “ticket” type= “tio:ticketOrder” />
    <element name= “seating” type= “tio:seatAssignments”/>
    <element name= “txId” type= “tio:chargeTxId” />
    <element name= “totalPrice" type= “tio:price"/>
  </all>
 </complexType>
</element>

</schema>
</types>

<!--============================================================ -->
<!-- messages externalized by airline and agent processes       -->
<!--============================================================ -->

<message name=“tripOrderMsg”>
  <part name=“order” element=”tio:tripOrder”/>
</message>

<message name=“tripOrderAck”>
  <part name=“agentWorkId” element=”wsfl:FlowInstanceId”/>
</message>

<message name=“itineraryMsg”>
  <part name=“itineraryInfo” element=“tio:itinerary”/>
</message>

<message name=“ticketOrderMsg” >
  <part name= “order” element=”tio:ticketOrder”/>
</message>

<message name=“ticketOrderAck” >
  <part name=“airlineWorkId” element=“wsfl:FlowInstanceId”/>
</message>

<message name=“confirmationMsg”>
  <part name=“confirmationInfo” element=“tio:confirmation”/>
</message>

<message name=“eTicketMsg” >
  <part name=“authorization” element=“tio:eTicket”/>
</message>

<!--============================================================ -->
<!-- messages used internally to define the signatures of       -->
<!-- activities in the airline business process                 -->
<!--============================================================ -->
```

```
<message name="receivedTicketOrder">
  <part name="request" element="tio:ticketOrder"/>
  <part name="airlineWorkId" element="wsfl:FlowInstanceId"/>
 </message>

<message name="reservation" >
   <part name="request" element="tio:ticketOrder" />
   <part name="airlineWorkId" element="wsfl:FlowInstanceId"/>
   <part name="seating" element="tio:seatAssignments"/>
   <part name="authorization" element="tio:eTicket" />
</message>

<message name="chargedReservation">
   <part name="confirmationInfo" element= "tio:confirmation"/>
   <part name="authorization" element= "tio:ETicket" />
</message>

<!--============================================================ -->
<!-- messages used internally to define the signatures of       -->
<!-- activities in the travel agent business process            -->
<!--============================================================ -->

<message name="receivedTripOrder">
   <part name="request" element="tio:tripOrder"/>
   <part name="agentWorkId" element="wsfl:FlowInstanceId"/>
</message>

<message name="tripRecord">
   <part name="request" element="tio:tripOrder" />
   <part name="agentWorkId" element="wsfl:FlowInstanceId"/>
   <part name="ourTicketOrder" element="tio:ticketOrder"/>
</message>

<message name="sentTicketOrder">
   <part name="orderRecord" element="tio:ticketOrderRecord"/>
</message>

<message name="confirmedTicketOrder">
   <part name="orderRecord" element="tio:ticketOrderRecord"/>
   <part name="confirmationInfo" element="tio:confirmation" />
</message>

</definitions>
```

8.3 Port Types Externalized by the Flow Models of the Travel Example

This section provides the definition of all the port types that are referenced in the travel agent and airline flow models. Each operation in these port types is defined in terms of its input and output messages; each of the messages used is a WSDL message as defined in the preceding subsection. If needed, import statements could be included to provide the actual locations where the schema or WSDL definitions can be found. As we mentioned before, the port type definitions presented in the following example will be assigned to the same target namespace as the message definitions of the previous section.

The standalone travel agent uses one port type for the operations he exposes to the traveler and a separate port type for interactions with the airline. Similarly, the standalone airline interfaces are split into separate port types for the interactions with agent and traveler. We

also provide the port type corresponding to the interface that a ticket buyer would need to implement in order to participate in the ticket-ordering process.

```
<definitions name="totalTravelPortTypes"
 targetNamespace=
 "http://www.TravelLuck.com/WebServices/Messages/TotalTravel"
 xmlns:tio=
 "http://www.TravelLuck.com/WebServices/Messages/TotalTravel"
 xmlns="">

<!--============================================================   -->
<!-- These are the standalone TravelAgent interfaces             -->
<!--============================================================   -->

<portType name="tripHandler">
   <operation name="receiveTripOrder">
      <input name="receiveTripOrderInput"
             message="tio:tripOrderMsg"/>
      <output name="receiveTripOrderOutput"
              message="tio:tripOrderAck"/>
   </operation>
   <operation name="sendItinerary">
      <output name="sendItineraryOutput"
              message="tio:itineraryMsg" />
   </operation>
</portType>

<portType name="ticketRequester">
   <operation name="requestTicketOrder">
      <output name="requestTicketOrderOutput"
              message="tio:ticketOrderMsg" />
      <input name="requestTicketOrderInput"
             message="tio:ticketOrderAck" />
   </operation>
   <operation name="waitForConfirmation">
      <input name="waitForConfirmationInput"
             message="tio:confirmationMsg" />
   </operation>
</portType>

<!--============================================================ -->
<!--This is the standalone Airline interface                    -->
<!--============================================================ -->

<portType name="ticketHandler">
  <operation name="receiveTicketOrder">
    <input name="receiveTicketOrderInput"
           message="tio:ticketOrderMsg"/>
    <output name="receiveTicketOrderOutput"
            message="tio:ticketOrderAck"/>
   </operation>
   <operation name="sendConfirmation">
       <ouput name="sendConfirmationOutput"
              message="tio:confirmationMsg"/>
   </operation>
</portType>

<portType name="ticketDelivery">
   <operation name="sendETicket">
       <output name="sendETicketOutput"
               message="tio:eTicketMsg"/>
```

```
   </operation>
</portType>

<!--============================================================  -->
<!--This is the interface required from a ticketBuyer           -->
<!--============================================================  -->

<portType name="ticketBuyer">
   <operation name="sendTripOrder">
      <output message="tio:tripOrderMsg"/>
      <input  message="tio:tripOrderAck"/>
    </operation>
    <operation name="receiveETicket">
        <input message="tio:eTicketMes"/>
    </operation>
    <operation name="receiveItinerary">
       <input message="tio:itineraryMsg"/>
    </operation>
</portType>

</definitions>
```

8.4 The Flow Models for Airline and Agent

In this section, we define the flow models that specify the standalone business processes for the airline issuing a ticket, as requested by a travel agent, and the travel agent contacting an airline in response to a traveler request to book a trip. For simplicity, all the definitions provided in this section will be presented as belonging to a WSFL document, although for reusability reasons, it might be convenient to separate them into several documents.

8.4.1 Service Provider Type Definitions

The definitions of the three service provider types used in the airline and agent flow models are provided in the following code, including, in particular, the service provider type that a ticket buyer would need to support to participate in the ticket-purchasing process. Each of the flow models would also need a "local" service provider type to provide the implementations for internal activities; we omit their definitions for brevity.

```
<definitions name="totalTravelPortTypes"
 targetNamespace=
 "http://www.TravelLuck.com/WebServices/Messages/TotalTravel"
 xmlns:tio=
 "http://www.TravelLuck.com/WebServices/Messages/TotalTravel"
 xmlns="">

<serviceProviderType name="airlineFlow">
   <portType name="tio:ticketHandler"/>
   <portType name="tio:ticketDelivery"/>
</serviceProviderType>

<serviceProviderType name="agentFlow">
   <portType name="tio:tripHandler"/>
   <portType name="tio:ticketRequester"/>
</serviceProviderType>

<serviceProvider name="travelerType">
   <portType name="tio:ticketBuyer"/>
</serviceProvider>
```

8.4.2 The Airline Flow Model

We are following the general naming convention of using “tickets” for objects relating to the airline, “trip” for objects relating to the travel agent and travel for objects relating to the traveler. Hence, this airline process becomes the `bookTickets` process supporting the `ticketHandler` and `ticketDelivery` port types.

We now introduce the service provider type `airlineFlow` to represent the airline business process. The definition of the `local` service provider referenced in the `performedBy` elements of some of the activities has been omitted as we did already before with the definitions of the corresponding service provider types.

```
<!--============================================================ -->
<!-- definition of bookTickets flow model                      -->
<!--         using airlineFlow serviceProviderType              -->
<!--============================================================ -->

<flowModel name="bookTickets"
                 serviceProviderType=”airlineFlow”>

<flowSource name=”ticketFlowSource”>
   <output name=”processInstanceData”
           message=”tio:receivedTicketOrder”/>
</flowSource>

<serviceProvider name=”agent” type=”agentFlow”/>

<serviceProvider name=”traveler” type=”travelerType”/>

<export lifecycleAction=“spawn”>
   <target portType=“tio:ticketHandler”
           operation=“receiveTicketOrder”>
      <map sourceMessage=“receiveTicketOrderInput”
           targetMessage=“processInstanceData”
           targetPart=“request”/>
      <map sourceMessage=“processInstanceData”
           sourcePart=“airlineWorkId”
           targetMessage=“receiveTicketOrderOutput”
           targetPart=“airlineWorkId”/>
   </target>
</export>

<activity name="reserveSeats">
   <input name=”reserveSeatsInput”
          message=”tio:receivedTicketOrder”/>
   <output name=”reserveSeatsOutput” message=”tio:reservation”/>
   <performedBy serviceProvider=“local”/>
   <implement>
     <internal>
     <!--  .. call to reservation system ..  -->
     <internal/>
   </implement>
</activity>

<activity name="chargeCreditCard" >
   <input name=”dataIn” message=”tio:reservation” />
   <output name=”dataOut” message=”tio:chargedReservation”/>
   <performedBy serviceProvider=“local”/>
```

```
   <implement>
     <internal>
     <!--  .. call to credit card service .. -->
     </internal>
   </implement>
</activity>

<activity name="confirmFlights" >
   <input  name=”confirmFlightsInput”
           message=”tio:chargedReservation”/>
   <output name=”confirmFlightsOutput” message=”tio:eTicketMsg”/>
   <performedBy serviceProvider=“agent”/>
   <implement>
     <export portType=“tio:ticketHandler”
             operation=“sendConfirmation”>
        <map sourceMessage=“confirmFlightsInput”
             sourcePart=“confirmationInfo”
             targetMessage=“sendConfirmationOutput”
             targetPart=“confirmationInfo”/>
     </export>
   </implement>
</activity>

<activity name="issueETicket">
   <input name=“issueETicketInput” message=”tio:eTicketMsg”/>
   <performedBy serviceProvider=“traveler”/>
   <implement>
     <export portType=“tio:deliverTickets” operation=“sendETicket”>
        <map sourceMessage=“issueETicketInput”
             sourcePart=“authorization”
             targetMessage=“sendETicketOutput”
             targetPart=“authorization”/>
      </export>
   </implement>
</activity>

<datalink   name=”gT-rS-data”
            source= “ticketFlowSource”
            target=”reserveSeats”/>
<controlLink
            name=”rS-cC”
            source=”reserveSeats”
            target=”chargeCreditCard”/>
<dataLink
            name=”rS-cCdata”
            source=”reserveSeats”
            target=”chargeCreditCard”/>
<controlLink
            name=”cC-cF”
            source=”chargeCreditCard”
            target=”confirmFlights”/>
<dataLink
            name=”cC-cFdata”
            source=”chargeCreditCard”
            target=”confirmFlights”/>
<controlLink
            name=”cF-sT”
            source=”confirmFlights”
            target=”issueETicket”/>
<dataLink
            name=”cF-sTdata”
```

```
            source="confirmFlights"
            target="issueETicket"/>

</flowModel>
```

In this example, we have exported the `spawn` lifecycle activity to the `receiveTicketOrder` operation. Thus, when this operation is called from some partner flow or service in a global model, a new instance of the airline business process represented by this `flowModel` is created. The `spawn` lifecycle operation creates a new process instance, assigns it a unique instance identifier of type `wsfl:FlowInstanceId` and returns to the caller immediately (without waiting for the created process instance to complete). Exporting the `spawn` process to a `receiveTicketOrder` operation allows application-specific initialization data to be passed in on the call namely the information specifying what ticket is requested–which is then made available through `flowSource` to activities of the process through data links.

The unique `FlowInstanceId` generated by `spawn` is passed back as reply data to the calling operation. In principle, this enables the agent requesting a ticket order to call in subsequently with other lifecycle operations to this process instance. The unique `FlowInstanceID` ID can also be used by the agent as a record locator for future messages associated with this ticket order. This includes correlation with confirmations from the airline and airline reference information to be printed on the itinerary for the traveler.

It is implicit in this example that the `wsfl:FlowInstanceId` value returned from the `spawn` operation is used to set the `airlineWorkId` part of the `flowSource processInstanceData` based on type matching.

8.4.3 The Travel Agent Flow Model

This section shows the standalone travel agent business process specifying just the travel agent processing independently of airline and traveler. Following our naming convention, this is the `bookTrip` flow model supporting the `tripHandler` and `ticketRequester` port types. The `serviceProviderType` defined for the agent business process is called `agentFlow`.

```
<!--=========================================================     -->
<!-- definition of bookTrip flow model                           -->
<!--         using agentFlow serviceProviderType                 -->
<!--=========================================================     -->

<flowModel name="bookTrip"
           serviceProviderType="agentFlow">

<flowSource name="tripFlowSource">
    <output name="processInstanceData"
            message="tio:receivedTripOrder"/>
</flowSource>

<serviceProvider name="airline" type="airlineFlow"/>

<serviceProvider name="traveler" type="travelerType"/>

<export lifecycleAction="spawn">
   <target portType="tio:tripHandler"
           operation="receiveTripOrder">
      <map sourceMessage="receiveTripOrderInput"
           targetMessage="processInstanceData"
           targetPart="request"/>
```

```
      <map sourceMessage=“processInstanceData”
           sourcePart=“agentWorkId”
           targetMessage=“receiveTripOrderOutput”
           targetPart=“agentWorkId”/>
   </target>
</export>

<activity name="selectLegs">
   <input name=”dataIn” message=”tio:receivedTripOrder”/>
   <output name=”dataOut” message=”tio:tripRecord”/>
   <providedBy serviceProvider=”local”/>
   <implement>
     <internal>
     <!--  .. start agent forms/dialog for select legs.. -->
     </internal>
   </implement>
</activity>

<activity name="orderTickets">
   <input name=”orderTicketsInput” message=”tio:tripRecord”/>
   <output name=”orderTicketsOutput” message=”tio:sentTicketOrder”/>
   <providedBy serviceProvider=”airline”/>
   <implement>
      <export portType=“tio:ticketRequester”
              operation=“requestTicketOrder”>
         <map sourceMessage=“OrderTicketsInput”
              sourcePart=“ourTicketorder”
              targetMessage=“requestTicketOrderOutput”/>
         <map sourceMessage=“requestTicketOrderInput”
              sourcePart=“theAirlineWorkId”
              targetMessage=“ordertickетsOutput”
              targetPart=“theAirlineWorkId”/>
      </export>
   </implement>
</activity>

<activity name="getConfirmation">
   <input name=”getConfirmationInput”
          message=”tio:confirmedTicketOrder”/>
   <output name=”getConfirmationOutput”
           message=”tio:confirmedTicketOrder”/>
   <providedBy serviceProvider=”airline”/>
   <implement>
     <export portType=“tio:ticketRequester”
             operation=“waitForConfirmation”>
       <map sourceMessage=“waitForConfirmationInput”
            sourcePart=“confirmationInfo”
            targetMessage=“getConfirmationInput”
            targetPart=“confirmationInfo”/>
     </export>
   </implement>
</activity>

<activity name="generateItinerary">
   <input name=”dataIn” message=”tio:confirmedTicketOrder”/>
   <output name=”dataOut” message=”tio:itineraryMsg”/>
   <providedBy serviceProvider=”local”/>
   <implement>
     <internal>
     <!-- .. start agent forms/dialog to compile itinerary.. -->
     </internal>
```

```
   </implement>
</activity>

<activity name="issueItinerary">
   <input name=”issueItineraryInput” message=“tio:itineraryMsg”/>
   <providedBy serviceProvider=”traveler”/>
   <implement>
     <export portType=“tripHandler” operation=“tio:sendItinerary”>
        <map sourceMessage=“issueItineraryInput”
             sourcePart=“itineraryInfo”
             targetMessage=“sendItineraryOutput”
             targetPart=“itineraryInfo”/>
     </export>
   </implement>
</activity>

<dataLink
            name=”gF-sLdata”
            source=”tripFlowSource”
            target=”selectLegs”/>
<controlLink
            name=”sL-oT”
            source=”selectLegs”
            target=”orderTickets”/>
<dataLink
            name=”sL-oTdata”
            source=”selectLegs”
            target=”orderTickets”/>
<controlLink
            name=”oT-rC”
            source=”orderTickets”
            target=”getItinerary”/>
<dataLink
            name=”oT-rCdata”
            source=”orderTickets”
            target=”getItinerary”
            <map sourceMessage=“orderTicketsOutput”
                 sourcePart=“orderRecord”
                 targetMessage=“getItineraryInput”
                 targetPart=“orderRecord”/>
</datalink>
<controlLink
            name=”gC-gI”
            source=”getConfirmation”
            target=”generateItinerary”/>
<dataLink
            name=”gC-gIdata”
            source=”getConfirmation”
            target=”generateItinerary”/>
<controlLink
            name=”gI-sI”
            source=”generateItinerary”
            target=”issueItinerary”/>
<dataLink
            name=”gI-sIdata”
            source=”generateItinerary”
            target=”issueItinerary”/>

</flowModel>

</definitions>
```

8.5 The Global Model **tripNTicket**

Having defined flow models for the agent and the airline, we now illustrate a composition referring to and using these flow models. It is the composition that, as a single service to the traveler, provides the combined services of travel agent and the selected airline.

A new port types will be externalized by this combined service. We show those as a WSDL file to be imported by the WSFL. Because there is now a single connection point between user and the combined `tripNTravel` service, a single port type is used for the service interface.

```
<definitions name=“totalTravelPortTypes“
 targetNamespace=
 "http://www.TravelLuck.com/WebServices/Messages/TotalTravel"
 xmlns:tio=
 “http://www.TravelLuck.com/WebServices/Messages/TotalTravel”
 xmlns=“”>

<!--=========================================================     -->
<!-- This is the working AgentNAirline porttype in WSDL           -->
<!--=========================================================     -->

<portType name=”tripNTicketHandler”>
   <operation name=“receiveTripOrder”>
      <input name=“receiveRequest” message=“tio:tripOrderMsg”/>
      <output name=“returnResponse” message=“tio:tripOrderAck”/>
   </operation>
   <operation name=“sendItinerary”>
      <output name=“sendMessage” message=“tio:Itinerary”/>
   </operation>
   <operation name=“sendETickets”>
       <input name=“sendMessage” message=“tio:ETickets”    />
   </operation>
</portType>
```

Next, we show the WSFL file for the composition that will begin with the definition of the new service provider type that uses and externalizes the port types defined above. This is immediately followed by the WSFL defining the composition that realizes this new service provider type.

```
<!--=========================================================     -->
<!-- definition of  agentNAirlineFlow  serviceProviderType        -->
<!--=========================================================     -->
<serviceProviderType name=”agentNAirlineFlow”>
     <portType name=”tio:tripNTicketHandler”>
</serviceProviderType>

<!--=========================================================     -->
<!-- definition of bookTripNTickets composition                   -->
<!--         using agentFlow serviceProviderType                  -->
<!--=========================================================     -->
<globalModel name="bookTripNTickets"
             serviceproviderType="agentNAirlineFlow”>

<serviceProvider name="travelAgent"
                 serviceProviderType=”tio:agentFlow”>
  <export>
```

```
    <source portType="tio:tripHandler"
            operation="sendItinerary"/>
    <target portType="tio:tripNTicketHandler"
            operation="sendItinerary"/>
  </export>
  <export>
    <source portType="tio:tripHandler"
            operation="receiveTripOrder"/>
    <target portType="tio:tripNTicketHandler"
            operation="receiveTripOrder"/>
  </export>
  <locator type="static"
          service="Traveluck.com"/>
</serviceProvider>

<serviceProvider name="airline"
                 serviceProviderType="tio:airlineFlow">
  <export>
    <source portType="tio:ticketDelivery"
            operation="sendETicket"/>
    <target portType="tio:tripNTicketHandler"
            operation="sendETickets"/>
  </export>
</serviceProvider>

<plugLink>
     <source serviceProvider="airline"
             portType="tio:ticketHandler"
             operation="sendConfirmation"/>
     <target serviceProvider="travelAgent"
             portType="tio:tripHandler"
             operation="waitForConfirmation"/>
</plugLink>

<plugLink>
     <source serviceProvider="travelAgent"
             portType="tio:tripHandler"
             operation="requestTicketOrder"/>
     <target serviceProvider="airline"
             portType="tio:ticketHandler"
             operation="receiveTicketOrder"/>
</plugLink>

</globalModel>

</definitions>
```

9 References

1. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, “Web Services Description Language (WSDL) version 1.0”, http://www.uddi.org/submissions.html, September 2000.
2. World Wide Web Consortium, “XML Schema Part 1: Structures”, W3C Candidate Recommendation, http://www.w3.org/TR/xmlschema-1/, October 2000.
3. World Wide Web Consortium, “XML Schema Part 2: Datatypes”, W3C Candidate Recommendation, http://www.w3.org/TR/xmlschema-2/, October 2000.

International Business Machines Corporation
Software Communications Department
Route 100, Building 1
Somers, NY 10589
U.S.A.

05-01