Implementing Interorganizational Workflows via Web Services Orchestration

C. W. Günther

Hasso Plattner Institute for Software Systems Engineering
Prof.-Dr.-Helmert-Str. 2-3, 14440 Potsdam, Germany
christian@deckfour.com

Abstract. This paper covers the subject of realizing complex business processes using the technique of web services composition. In the first part the basics of web services orchestration are introduced using IBM's proposed WSFL standard both as notation and as an implementation example. Both the flow model, defining interaction modalities and constraints of the composed web services, and the global model, describing interfaces and connection of multiple independent service providers, are introduced with their dominant elements. The second part deals with the technique of designing distributed workflows using the Public-To-Private (P2P) approach, which allows for defining stable interfaces and assigning responsibilities while guaranteeing invariant overall flow semantics. After a transformation of the P2P model into WSFL notation the interactions of this combination are discussed, including implications on the field of networked environment business. To prove the concept of interorganizational workflow implementation based on web services a prototypical flow engine for WSFL has been developed whose architecture and implementation is described in the third part. The last part finally discusses implications, the conjunction of web service orchestration and the P2P approach has on some aspects of business process modeling prospects.

1 Introduction

In the internet as we know it for today, the notion of distributed client-server computing is almost completely limited to HTML forms interacting with dynamic database-driven web sites. Web services are about to change the way we think about dynamics regarding the net, introducing the means to provide services worldwide using a set of common standards, rarely seen in networked application domains lately. Yet, this brave new world of ultimate interoperability lacks one crucial aspect necessary for complex applications: web services are stateless operations, thus only providing for small, atomic applications like weather services or the like.
In the business world people have long since discovered that keeping the overview in complex, yet standardized, processes is often better left to machines putting small steps together and supervising large business processes' execution, which takes also place in small, atomic actions. As in today's networked economy featuring highly dynamic market environments, enterprises of most different backgrounds have to collaborate, they are in a bad need for some agreed upon standards for using each other's services. Web services are about to bring that long awaited standardization, at least in the B2B sector, yet the web services stack still lacks support for complex business processes. This is a field which is, at the moment, subject to busy developments: Web services orchestration, also known as web services choreography or composition [Leymann, 2001], will bring the necessary process support for web services, overcoming stateless calls and at last providing for dynamic workflows crossing company borders and being even independent from specific enterprises' participation.
This paper presents, as an available technique for web service orchestration, IBM's proposed standard for WSFL, the Web Services Flow Language. Further, implications, the combined use of workflow techniques and web service technologies in the form of web service orchestration has on the way that enterprises will collaborate in the future, and on the entire business environment, will be discussed, presenting a translation and further adoption of the Public-To-Private (P2P) Approach to Interorganizational Workflows [van der Aalst and Weske, 2001] to WSFL-based web services orchestration.
Today, the composition of multiple web services to one more abstract service is, even more than the overall use of web services technology, merely a vision we are heading towards. No implementation of a web services choreography framework can be found, as standardization in this field is still in progress. For this cause, as a proof of concept, a simple and limited framework of a WSFL-based web services flow engine has been developed, which is yet able to perform the basic tasks needed to implement interorganizational workflows based on web services. This prototypical system is introduced in part five of this paper, explaining how an implementation of such an engine could be developed and making things clearer by shifting theory to praxis.
At last a discussion of some open issues regarding the whole notion of interorganizational workflows in conjunction with web services is presented, reshaping the big picture of this emerging technology.

2 Basics

To understand the crucial matters that arise, when applying workflow techniques to the field of web services, it is necessary to have a basic understanding of both subjects. Although this paper strives not to concentrate on implementational details and to keep to aspects relevant for the actual focus, one has to obtain at least a basic understanding of the theoretical and technological foundations to perceive the key topics and comprehend arising implications.
The technology of web services composition is layered on top of the web services stack, making use of various, if not all, included standards. As for the WSFL language, heavily based on the Web Services Description Language (WSDL), invocation standards like SOAP (Simple Object Access Protocol) and XML-RPC are taken for granted, like lookup devices as UDDI (Universal Description, Discovery and Integration) are necessary to make workflows dynamic with respect to the participants. Given the fact, that this paper is preceded by various contributions covering all of these necessary aspects in detail, one will most surely encounter no difficulties in getting things together.
Workflows are constructs describing the interaction of single, atomic business activities within a larger, composite business process, thus allowing for both top-down analysis of complex business tasks into smaller compartments and, the other way round, bottom-up design from small modules to large business processes. As for the web services part I presuppose your being familiar with this topic from previous contributions, please refer to these for uncertainties possibly arising. The implementational focus of this paper, the WSFL language, is a workflow language which differs but slightly from traditional languages in this field. In this context, all aspects of workflow design, like workflow patterns, can be easily transformed into WSFL.

3 The WSFL Language

WSFL, the Web Services Flow Language, is a standard proposal for web services orchestration by IBM, currently available in its premier draft version 1.0 [Leymann, 2001]. It is, from an implementational point of view, an XML language heavily based on WSDL.
The WSFL specification considers two types of web services composition which are represented by their respective different model types.

3.1 The Flow Model

The first type of web services composition specifies a usage pattern of multiple web services, orchestrated in such a way that the resulting model describes how to achieve a particular business goal, i.e. the description of a business process [Leymann, 2001]. This results typically in the specification of a workflow, in which we have activities represented mainly by web services, besides control and data links, defining the execution order, alternative branching policies and, the latter, data flow between the single activities making up the flow.
Flow models allow for the creation of workflows or business processes consisting of web services as they provide, through control and data links, the means necessary to specify execution control and exact data mapping between the single activities involved in the flow. As a graphical representation of flow models WSFL introduces workflow graphs, directed graphs consisting of activities connected by the respective control and data links as edges.

3.1.1 Activities

When a company is about to model a specific workflow, the first thing to do is usually identifying the involved operations. In WSFL these operations are identified as activities, which are represented by circles as nodes in the workflow graphs.
The nature of activities is to describe abstract steps used to achieve the overall business goal, the actual work is carried out by their specified operations. Notice, that activities identify business tasks which can be either single-step operations like web services, or represent a composite business process like another workflow itself, thus emphasizing the recursive character of WSFL flow models.
Activities have a signature related to their specific implementation, i.e. they can have an input and output message used to retrieve the data necessary to execute the implementing operation and to store output data.
The *implementation* part in activity specification describes the binding of an actual implementing operation to the activity, which can either be provided locally, using the *internal* element, or accessed remotely, then specified using an *export* element. When an activity completes, the success of its execution may determine further flow branching, for this case WSFL features exit conditions to determine the successful execution of activities. The other way round, an activity may specify through join conditions, Boolean expressions operating on the value of input control links, the prerequisites for its execution. The default behavior for join conditions is a non-exclusive OR, i.e. any control link terminating at the activity has to evaluate to true; for exit conditions the default behavior is to terminate with success, when the actual success of the implementing operation cannot, or is not specified how to, be determined.

WSFL code sample:

```
<flowModel name=”bookLover” serviceProviderType=”bookLoverPublic”>
(...)
    <activity name=”selectBook” exitCondition=”bookDictionary.status='OK'”>
        <input message=”bookOrder”/>
        <output message=”bookDictionary”/>
        <performedBy serviceProvider=”bookseller01”/>
        <join condition=”POaccepted AND SRreceived” when=”deferred”/>
        <implement>
            <export>
                <target portType=”bookRequester” operation=”orderDictionary”/>
            </export>
        </implement>
    </activity>
(...)
</flowModel>
```

3.1.2 Control Links

The second step in modeling workflows is to identify the rules, after which the single activity steps can be sequenced. This is, on the one hand, defining the order in which these activities can follow one another. Another aspect is a conditional transition from one activity to another. Both of them are represented by control links in WSFL.
By specifying source and target activity of a control link the basic means for an ordered sequence of execution are established, further control is provided by the use of transition conditions. These are basically Boolean expressions operating on actual and formal parameters, where the latter can be part of any message produced earlier in the flow. As WSFL allows for specifying a transition condition for every control link, this low-level control structure makes it possible to model the flow sequence control in a very fine-grained, though merely uncomfortable manner.
When no transition condition is specified for a given control link, this control link evaluates to true by default, i.e. the target activity is always executed, assuming the source activity has been executed successfully.
This mechanism of low-level conditional branching spawns the whole dynamics of business process dynamics, as it becomes possible to model flows in a way very similar to conventional programming, thus allowing for highly dynamic workflows. Control links are the directed, weighed edges of the workflow graphs, and can be followed only once in execution, i.e. the graph is directed and acyclic, loops are forbidden (*with the exception of do-until loops based on the fact, that execution of an activity is, by default, repeated until it evaluates to true: put this way, activities implemented by flows themselves combined with appropriate exit conditions can be used to iterate the activity-contained flow*).

WSFL code sample:

```
<controlLink name=”sub-ship-1”
    source=”processPO”
    target=”acceptSR”
    transitionCondition=”processPOOutput/x &gt; acceptSRInput.y”/>
```

3.1.3 Data Links

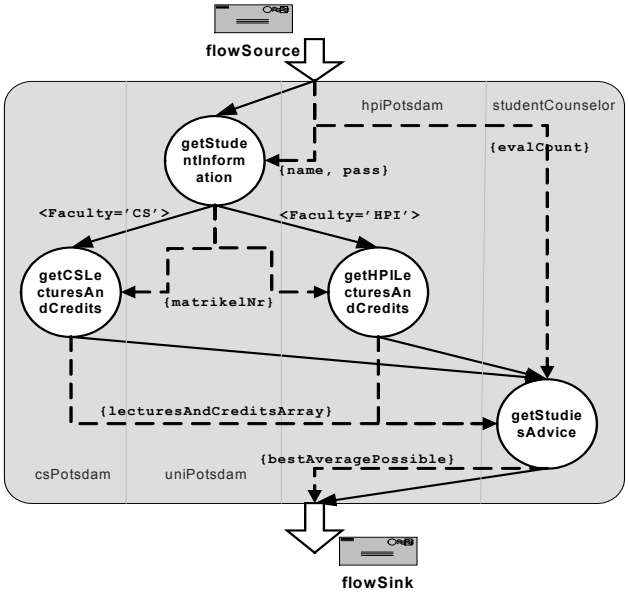
Describing the flow of information between activities is the third step in creating a workflow, the question is, what part of which activity's output is used as input for another activity. These data flows make up the second kind of directed edges in workflow graphs, the data links.
A data link specifies a defined part of the source activity's output data to be passed to the target activity's input data, allowing activities to use data created by any activity previously executed in the flow. As it must always be ensured that the source data has already been created when the target activity is about to be executed, the limitation is set up that data links must only connect two activities, where the target activity can be reached by following a path of directed control links from the source activity (i.e., *data flows along control streams*).
Data links can be weighed by map specifications, allowing not only for simply passing data message-wise, but also for setting up more complex data flows, where one activity's input message is composed of multiple activities' output data or, the other way round, passing different parts of an activity's output data to different receiving activities. Map specifications simply define how message parts are to be mapped to other message parts.

WSFL code sample:

```
<dataLink name=”flowModel-ship” source=”flowSource” target=”acceptRequest”>
    <map sourceMessage=”anINVandSR” targetMessage=”anSR”
        sourcePart=”SR” targetPart=”SR”/>
</dataLink>
```

3.1.4 Workflow Graphs

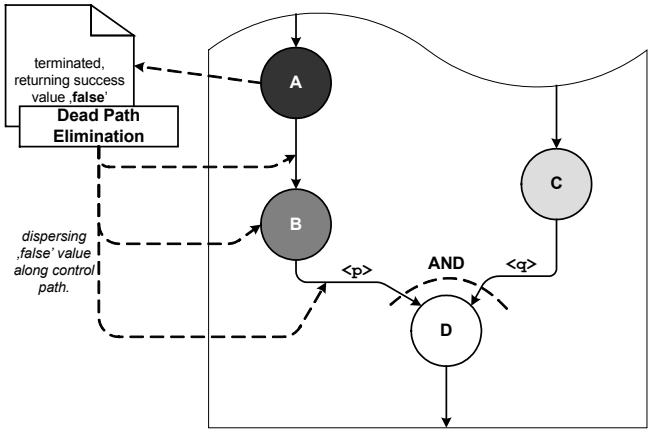
As WSFL flow model documents do not reflect the structure of the flow in their internal structure they allow only for limited human readability. Modeling workflows with directed graphs is considered very useful long since, and WSFL makes no exception proposing workflow graphs as graphical representation for flow models.
As stressed in the discussion of the major flow model elements earlier, activities are represented by nodes, depicted in the graph by circles. The edges of the directed graph are both control and data links, with control links depicted by conventional lines and data links by dashed lines.



[Illustration 1: Flow model of the student counselor example]

Illustration 1 features the Student Counselor example as familiar from earlier contributions, here modeled as a workflow featuring four activities, each carried out by a dedicated service provider.
A flow can, as depicted in illustration 1, have overall flow input and output messages, defined by **`flowSource`** and **`flowSink`** elements. The flow starts with passing control to the *getStudent-Information* activity, carried out by service provider *uniPotsdam*, which also gets passed student name and password via a data link. One part of the activity output, *faculty*, is used for branching by testing it for equality with 'HPI' or 'CS' in the respective following control links' transition conditions. The other part, *matrikelNr*, is passed via data links to both following activities. Notice that the exclusive alternative utilized here is not implemented by using high-level constructs, but merely by encoding this 'exclusive OR' branch into the individual following control links' transition conditions. The rest of the flow is made up in a similar manner, and, as workflow graphs provide for intuitive access to flow semantics, will not be covered into deep at this place.

While executing the flow, the engine is following control links, executing all activities along the control path. As activities can have an exit condition determining that they have not been terminated successfully, and control links can evaluate to false, there has to be a mechanism passing the respective 'false' token along these paths. This mechanism is specified in the WSFL standard as Dead Path Elimination.



[Illustration 2: Dead Path Elimination]

Take the example situation depicted in illustration 2, showing part of a flow graph: Activity *C* has completed successfully while activity *A* has returned a success value of 'false'. The problem is in the question, how *D* can 'know' if it is about to be executed or not, as it has an 'AND' type join condition demanding 'true' condition values both from control links originating at *C* and at *B*. With the control link from *C* the case is clear, as *C* was executed successfully it will transport a 'true' value. But at the left path, execution has stopped with *A*, as this activity has returned 'false', and the control path is therefore not followed anymore in execution, preventing any success value being passed to *D* from this path. For such cases it is necessary to have Dead Path Elimination. This mechanism takes action whenever a flow element has been evaluated to 'false', traversing the path along and marking all path elements, i.e. in this case activity *B* and both control links from and to *B*, as evaluated and assigning them a 'false' value. Notice that this dead path elimination stops at activity *D*, as now the further execution is dependent on the join condition of *D*, which will, caused by the 'false' value from the left path, also evaluate to false. Now, dead path elimination will start again with the control link leaving *D*, following the further path and marking all following elements 'false'.
Another situation that requires the use of dead path elimination is quite similar to the example. Assuming the two control paths depicted above originated at a previous activity where the control path was split into these two paths executed concurrently and *D* was a synchronization node with an 'OR' type join condition. If no dead path elimination mechanism was carried out to make the fact known to *D*, that the left path will never be executed because of an earlier false exit condition, this activity had to wait forever, leaving the flow in an erroneous state.

3.1.6 Lifecycle Interface

Flow execution can be controlled remotely, i.e. explicitly from outside the flow itself, by the lifecycle interface operations accessible through port types implemented by each flow. The operation *spawn* triggers the execution of a flow, returning a system-unique flow ID used in calling further lifecycle operations.
The available set of lifecycle operations is providing an observer concept, enabling responsible authorities to gain information and control about flow execution, which itself is running autonomously.
In the following a short overview on flow lifecycle operations is provided.

- **`spawn`** – This is the premier operation in flow lifecycle control, creating a new flow instance and immediately starting its execution. The `spawn` operation returns the unique ID of the flow spawned.

- **`call`** – This operation is similar to the spawn operation, yet it is only returning after the flow spawned has been completely executed, returning the flow output message as result.

- **`suspend`** – Suspends flow execution, i.e. flow execution is interrupted until further lifecycle operations change this state. When a flow enters suspend state, all activities currently executing are terminated and the current state of execution is remembered.

- **`resume`** – A previously interrupted flow can be resumed with this command, i.e. its execution is continued. The flow resumes execution exactly at the point it has been previously interrupted, using saved execution state for flow reconstruction.

- **`enquire`** – This operation returns the current status of a flow without interrupting or resuming it.

- **`terminate`** – When terminate is called the execution of a flow is immediately terminated and all flow data is deleted.

As an example for the WSFL syntax regarding lifecycle operations, a sample specification of the *spawn* operation is provided:

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

3.2 The Global Model

For modeling web services as a composition of existing ones, WSFL provides the global model, which is closely linked to the notion of a recursive composition metamodel. The global model formally describes how different service providers are linked together from an architectural point of view, i.e. it focuses more on the interfaces and their connections but on the sequencing order, which is the domain of flow models.

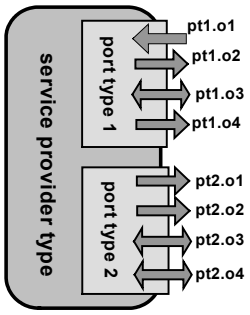
3.2.1 Global Models

Global models are WSFL's means to define, in a simple manner, interactions between service providers and the composition of new services from existing ones. From the global model point of view a web service implemented by a business process as a flow model does not differ from an atomic web service. The interface of a web service is defined by a service provider type, which consists of a collection of port types presenting the operations making up the web service. Service provider types are the atomic elements of both flow models and global models, where both present different but complementary points of view on the same subject.
Flow models are made up of activities, which are bound to actual service providers providing the implementing operations. Activities therefore define requirements towards the service providers, which are actually chosen based on defined Locator elements, choosing one service provider implementing the required external interface following a specified algorithm. Locators can be static, i.e. they bind activities statically to certain providers, which is useful when it is sure that only this provider is to be used. But they can as well define the service provider to be determined dynamically, by either defining the locator to be of type '*UDDI*', which causes the flow engine to search a UDDI database for a matching service and chooses an appropriate one following a determined algorithm. Another dynamic

locator type is '*mobility*', which causes the engine to look up a matching service based on the current location of any client device.
Where flow models define execution sequence of activities and the data flow between them, the global model focuses on modeling the interactions between multiple service provider types, represented by plug links connecting dual operations on two different service provider types. These operations are dual in the sense of, that e.g. a notification operation on one service provider is bound to a one-way operation on another service provider, i.e. the call mechanisms have to match.
As for the relation between flow models and global models, keep in mind that the first ones describe the internal structure of (composite) web services while the latter define interactions between web services.

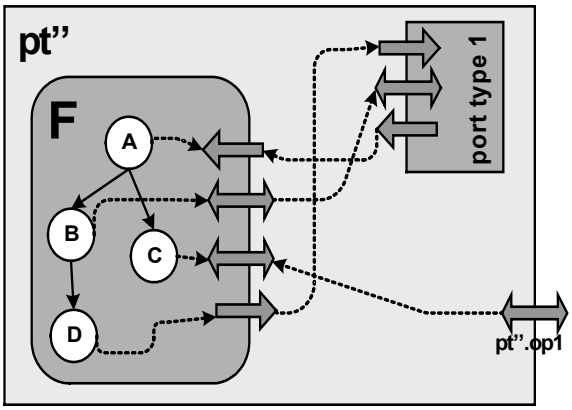
3.2.2 Service Providers

Basically, service providers are a collection of port types, defining the external interface of a flow. “Service providers are the units, from which global models are built” [Leymann, 2001], they can be perceived as peer-to-peer partners connected to each other to form one unit of a greater stage of abstraction by a global model. Actually, service providers can either represent the public interface of a flow, the interface of a web service or anything of the like, and their connection can create another service provider, which can in turn be used in more abstract models, underlining once more the recursive character of WSFL models.
The connection of service providers to one another is implemented by the binding of dual operations from both providers: Solicit-response operations can be connected to request-response operations and notification operations to one-way operations, where the connection between such two dual operations is described by the plug link construct in WSFL.
Plug links are no more but the representation of such an operation binding between two service providers in a global model, and can be perceived as event/message propagation channels or as connections between a client and a server operation, depending on your favorite paradigm.
As the signatures of dual operations are assumed to often not match completely, and as the need may arise for defining activities with signatures different to those of their implementing operations, both the export and the plug link construct allow for the specification of internal data mappings. These map constructs are in many ways identical with the mapping constructs known to be contained within data links, but are in their effect limited to data exchange between the involved activities and operations.
Service provider types can also be constituted by flow models, providing internal implementation for the external interface, where from a global model point of view the difference between a monolithic service provider and a flow model interface is irrelevant. Service provider types symbolizing flow models include at least one port type providing the flow's lifecycle operations, additional port types can be used to define the flow's requirements towards other business processes used in flow implementation. For each activity implemented by an external web service, the **`export`** element defines an association between activity and the implementing operation bound to the flow's service provider port, specifying the requirements of the activity for the actual implementation. The actual connection between activity and implementing web service is established by the global model's plug link defining the interaction.



[Illustration 3: Service Provider Type]

The graphical representation of port types and service providers is depicted in illustration 3. Operations are represented by a single arrow each, the direction of the arrow identifying the type of operation: An arrow pointing to the inside, e.g. pt1.o1, identifies a one-way operation, i.e. the supporting endpoint receives a message; an arrow pointing to the outside, e.g. pt1.o2, identifies a notification operation, i.e. the supporting endpoint sends a message. Dual-headed arrows identify either solicit-response (endpoint sends a message and expects correlated return message) or request-response (endpoint receives a message and sends return message) operations. Usually dual-headed operations are represented by horizontally shaded arrows, the dark-shaded head representing the direction of the first action.
Port types are named collections of operations, and respectively service provider types are constituted of a named collection of port types. Actual instances, i.e. service providers and ports, are illustrated the same way as their abstract types.



[Illustration 4: Global Model with transparent and opaque port types]

Activities are bound to their port operations by specifying an export element, graphically depicted by a dotted arrow pointing from the activity to the port operation.

Plug links are symbolized the same way, the arrow is here pointing into the direction of the first call stimulation. Notice that, like in illustration 4, global models can include both port types representing an internal implementation by a flow model (e.g. *F*) and opaque port types (e.g. *port type1*). The latter are specified only by their exported interface; the global model is insensitive for implementation details as long as the interface matches the given requirements.

3.2.3 WSFL syntax

The following example is to exemplarily introduce the syntax of global model language elements in WSFL. Notice, that the service provider and the adjacent locator element, as well as the export statement, appear as well in WSFL flow models, underlining the common interface of both models and their complementary nature.

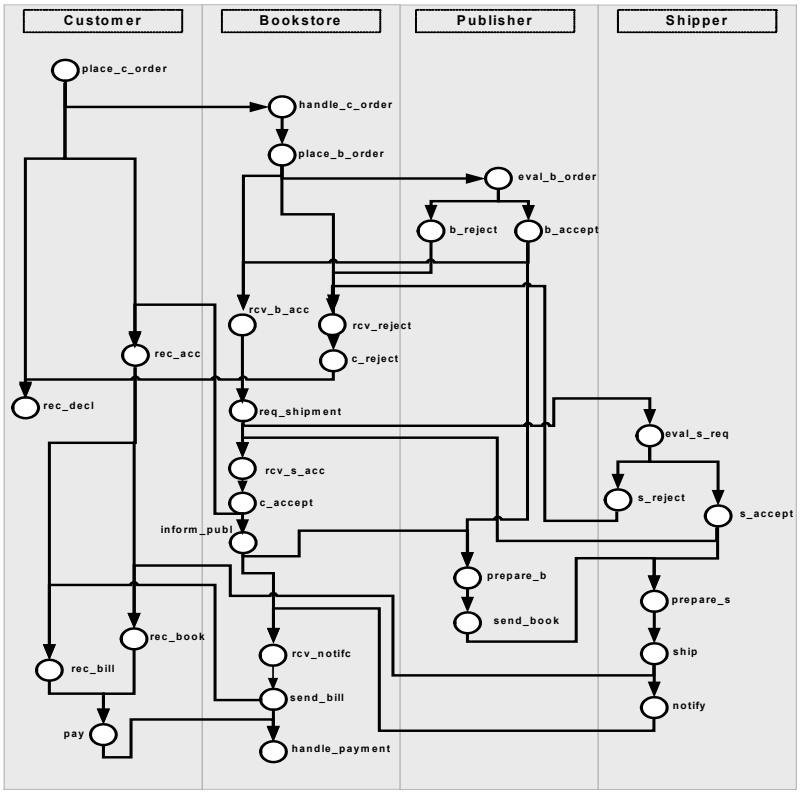
```
<globalModel name="orderingSomeBooks" serviceProviderType="compoundBookOrder">

    <serviceProvider name="bookseller01" 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>
    (...)
</globalModel>
```

4 WSFL and the P2P Approach

The Public-To-Private (P2P) approach, as known from a previous contribution, is a technique enabling workflow architects to create flexible interorganizational business process models. By first defining a public model including all participating organizations' main activities and then partitioning this public model into domain specific components, sort of a contract is introduced, agreed upon by all participating organizations. Based on this model, the common interfaces of all process participants can be defined, which paves the way for standardizing complex business process types involving multiple service providers.
The technique of orchestrating web services by means like WSFL flows can be used to analyze interorganizational cooperation possibilities and to create and implement both global and private parts of P2P models.

4.1 Using WSFL in the development of P2P Models - The Bookstore Example Revisited

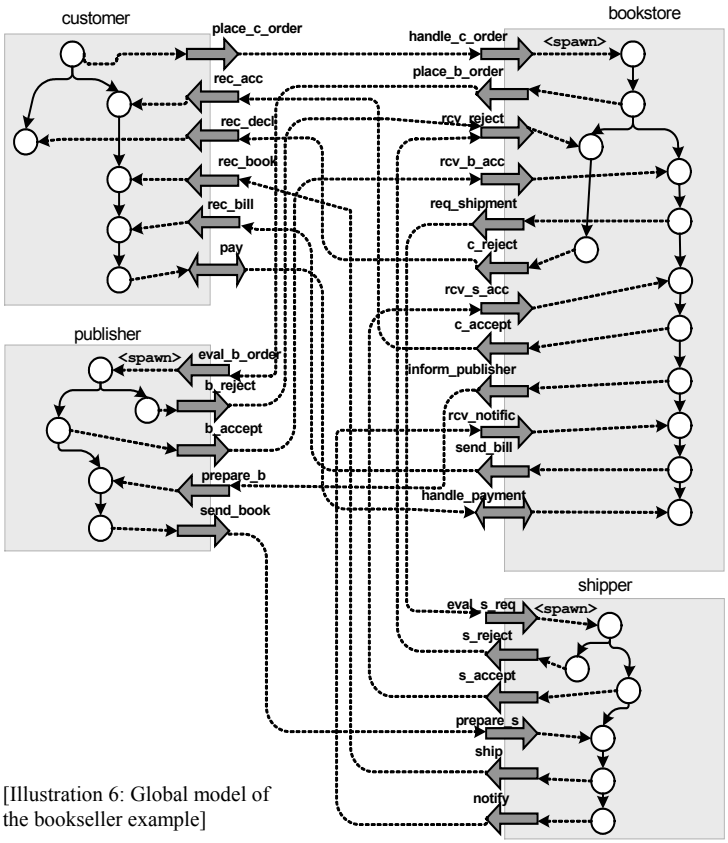
For describing how the P2P approach is translated into means of WSFL, this paper refers to the bookstore example presented in [van der Aalst and Weske, 2001] . The public workflow is in this case already presented in workflow net format, which can be translated into WSFL's workflow graphs with little effort.



[Illustration 5: Public flow model of the bookseller example]

The original public workflow had to be altered in mainly two points to make it applicable for WSFL: The WSFL specification does not allow for internal loops, but the original workflow contained two loops for both publisher and shipper determination. To preserve compliance with WSFL specification the iterated parts have either to be packed within a separate flow, called as an activity with an exit condition realizing a 'do-until' loop, or the loops have to be removed from the flow. For this exemplary examination I have decided to decline business process execution on any occurrence of the publisher or shipper not being able to serve a request; this simplification of the process should be justified by the overall public model becoming either too abstract or way too complex when performing a full loop translation.

The notation presented here (Illustration 5), a workflow graph with the contained activities already partitioned into 'swim-lanes', according to their specific domain, is considered most suitable. It both keeps the notion of a 'big picture', a common ground for communication about the overall business process, and it allows for easy classification of single activities, as both their connections and assignment to one of the participating domains can be intuitively perceived.
Such a public flow model is most likely to be the first document produced in the development of an interorganizational business process; the organization being the driving force among the participants or a joint committee can be imagined to evaluate given alternatives and, at last, present this model to the participants for them to agree upon.

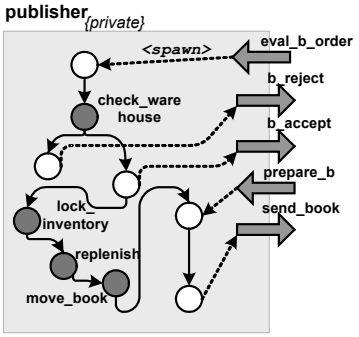


[Illustration 6: Global model of the bookseller example]

The next step would be splitting this flow model into domain-specific public parts, featuring the design of interconnections and interfaces between the involved partners. Such a description of public flow interfaces and the layout of actually 'wiring' the participating service providers together with regard to a specific business process, is calling for the usage of WSFL global models. These can be used in conjunction with numerous flow models describing the public parts of the participants and the bindings of their activities to publicly provided port interfaces.
The illustration presented here (illustration 6) is a combination of the four public parts as flow models and the overall global model of the bookseller example process. All acti vities are bound to the respective flow's port operations, while these operation ports are interconnected via plug links, describing which operation is actually called by whom. As such a graph is considered not suitable for actually depicting composite business processes with a high degree of distribution,

it can serve as a shared documentation of all involved interfaces and their wiring together, featuring opaque service providers for an increased readability.

With this plan in mind, each partner involved in the interorganizational workflow can just cut out their relevant part, including public part flow model and service provider and port type specification, and complete development and implementation of the actual private workflow component without having to consult the other partners anymore. Designing the private part of the workflow requires all participants to keep to the interface and port definitions given in the global model, thus preserving interoperability among the partners in compliance to the protocol agreed upon. Further, when expanding and altering the private part of the overall workflow, all participants have to keep to the rules and guidelines given in [van der Aalst and Weske, 2001] to make sure, the outcome represents a subclass of the public part. This is absolutely necessary to guarantee continuous semantics of the overall flow, as it is taken for granted by both the other participants and users of the business process, that the private parts implemented by the single partners represent the semantics of their respective private part, which is not to be subject to change in any way.



[Illustration 7: Private part]

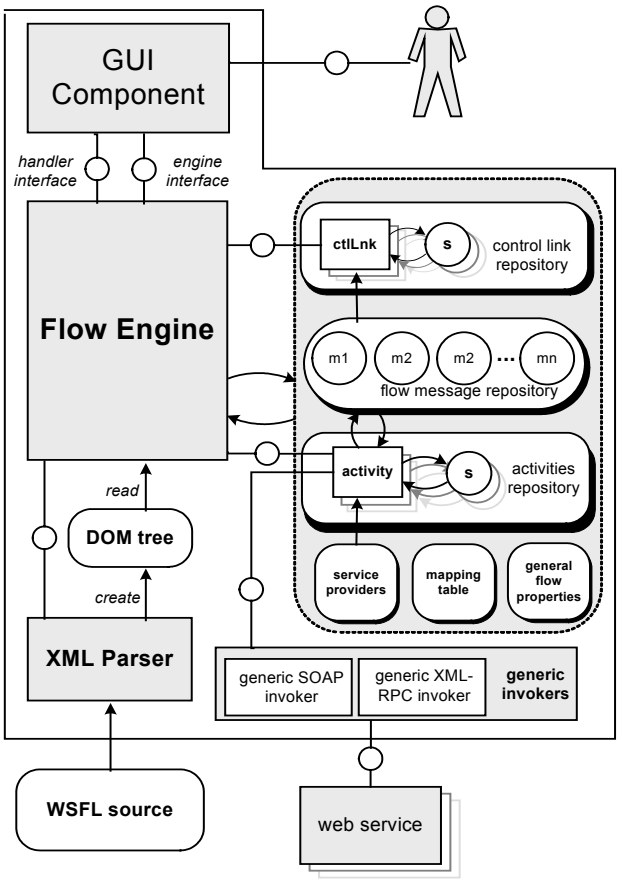
One possible specification of a private workflow part is presented in illustration 7, showing the private part of the publisher domain as exemplarily designed in [van der Aalst and Weske, 2001], translated into a WSFL flow model. The service provider and port type description part is merely for illustrating that this private implementation is still subject to the interface specifications defined in the public global model, likewise the actual flow is a subclass of the public part defined for the publisher domain. The newly added activities, here presented in gray color, are merely inserted into activity sequences which keeps conformity with the public part regarding the semantic of the private and overall flow.
Putting it all together, WSFL does not only present the means for developing and describing both public and private workflows of distributed business processes but extends the possibilities of P2P documentation. The use of global models featuring transparent service provider types allows for a combined presentation of both dynamic flow properties and a global interface and interconnection scheme, which is especially useful for communication about these aspects and enables developers at the participating organizations to see their part with respect to the overall public flow.

4.2 Combining P2P and Web Service Orchestration - Steps Ahead

The P2P approach is trying to develop defined responsibilities in a concerted business process development and to enable the participant parties to design a common base, the public flow. The latter is for defining interfaces and interconnection schemes for a standardized collaboration in the given business context.
On the other hand we have the powerful new means of web service orchestration, envisioned to ease both the combination of single web services to a new, single web service, and to ease interorganizational collaboration by defining meta-standards for information exchange and service usage across company boundaries. The great opportunity, web services are bringing to the world of B2B cooperation, and that is the domain, the P2P approach addresses, is a standardization by introducing flexible meta-standards. Web service protocols help loosen the coupling between business partners and make it possible to combine services from quite different technological backgrounds, as they do not assume one single implementation technology.
Web service orchestration, by means like WSFL, offers a powerful and, most important, standardized framework which complex distributed business processes can be built upon. Business partners can agree upon collaborating via the web services stack, using WSFL for developing the choreography of their joint business process. The design of this interorganizational workflow can then be realized following the P2P approach, which eases the distribution of responsibilities among the collaborating partners and creates an abstract architectural framework for the business process to be developed upon.
But notice, that the technique of web service orchestration has more to offer to the field of interorganizational workflows than just a set of basic technologies to build joint applications from. New, previously unknown elements in the field of B2B collaboration, like dynamic locators allowing these dynamic workflows to exchange business partners with the snip of a finger and completely without administration and user interaction, allow for more than just collaborations between single enterprises. It could be envisioned that an environment will arise, where interaction schemes are developed, defining collaborations between *classes* of organizations, with the participating partners being completely exchangeable.
It is a justified expectation, that the set of web services technologies, including techniques like WSFL for large-scale choreography, will in fact create common standards in B2B collaboration, making the development of custom solutions or acquisition of specialized solutions virtually obsolete. The market entry barriers for smaller enterprises and start-up companies will be significantly lowered which is most sure to further boost market dynamics and create a highly volatile business environment.

5 Proof Of Concept – A prototypical WSFL engine implementation

In its current state, the WSFL standard is merely a proposition, and as such subject to constant change and further development. The public of web services composition developers around the W3C and IBM's alphaworks division are lively discussing WSFL properties, proposing enhancements and customizations. This leads to the fact that, for the vast majority of developers and researchers not involved in IBM's WSFL workgroup, there is no system available which is actually able to perform WSFL flow execution.
As this paper should not only deal with the theoretical aspects of composing web service flows it was decided to implement an experimental flow engine based on the WSFL 1.0 draft specification [fLowRider].



[Illustration 8: Architecture of the prototypical flow engine]

5.1 Flow Engine Architecture

The flow engine was developed in Java™, using a strict object-oriented architecture. The main implementation of flow logic is located at a class *WSFLEngine*, which is controlled by a graphical user interface class *WSFLFlowRiderMain*. The engine class holds various hash maps and vectors containing references to all flow-defining elements such as activities, control links, and messages. All flow defining elements have been implemented as separate classes, their respective data contained as member attributes and main functionality implemented inside member methods. A block diagram, following the FMC notation presented in [Keller et al., 2001], is provided in Illustration 8 as a guidance for understanding overall system architecture.

The system relies on the Apache Xerces XML Parser in reading WSFL source documents and extracts the relevant information obtained using the DOM interface. For the actual invocation of web services operations implementing activities, the flow engine relies on two generic invocation classes developed by Sebastian H. Schenk, providing an intuitive interface for the invocation of both SOAP and XML-RPC based web services and using vectors and object arrays for data exchange.

5.1.1 Activities

Activities hold various attribute members defining their name, adjacent service provider and port type, the implementing operation and input and output message names. They implement one single method *run( )*, which is actually performing activity execution.
The *run* method first gets hold of connection information from its adjacent service provider and references to input and output message by calling respective engine methods. The data parts from the source message are obtained in their defined order in a vector, which is subsequently passed to either the *GenericXmlRpcInvoker* or the *GenericSoapInvoker*, based on the protocol type defined in the connection information.
The data returned from the generic invoker in an object array or vector is then, via engine method calls, inserted into the output message of the activity following once again the defined order of contained message parts.

5.1.2 Messages

The message class defines as attributes the message name, a hash map containing the message part objects mapped to their respective name and a supplementary vector containing the part names in their defined order. Methods implemented by the message class include extended get and set methods for all attributes, a size inquiry method and additional methods for either retrieving the part data in an ordered vector or inserting such an ordered vector in the correct manner.

5.1.3 Service Providers

Attributes defined in the service provider class include the service provider name, type, adjacent locator type and service and hash maps for mapping operation specific protocol type and URN to operations provided by the service provider. Methods include an advanced set of get and set methods and a modular set of operations for progressive composition of the service provider attributes by the engine, such as adding operations during initialization.

5.1.4 Control Links

Control links hold as attributes their name, source and target activity, a Boolean value indicating if they have a transition condition on them, the transition condition operation (such as 'equals') as string and a string array containing the operands necessary for transition condition evaluation.
Methods implemented by the control link class include, besides basic get and set methods, two comparison methods allowing to check for control link source and target equality with an argument string, a method determining the existence of a transition condition and the major method for evaluating the possibly adjacent transition condition.
The latter method first checks for the presence of a transition condition, if none is present it returns true by default. If a transition condition is present it resolves the operands, static type operands are used on an 'as-is' basis, i.e. they represent static values that do not have to be resolved. Otherwise the defined message part content is obtained from the engine repository and assigned to the operand. Subsequently the defined comparison method is performed on the operands and the resulting Boolean value is returned.
At the present state the control link class supports only the '*equals*' comparison operation, testing two operands for equality, but the overall design of transition condition evaluation allows for an easy extension by merely adding new comparison constructs.

5.1.5 Flow Engine

The flow engine class is the central component of the implementation, controlling flow creation from WSFL source files and subsequent execution. As it is implemented as a singleton a central *engine()* method provides engine access for all element instances constituting the flow. The engine holds numerous hash maps and vectors containing the flow element objects mapped to their names and additional attributes defining start and end activities, flow properties such as the flow name and execution state variables.
The method interface of the flow engine class provides basically three kinds of methods: The first kind deals with flow creation, the second kind provides access to flow elements managed by the engine and the third kind provides an interface for flow execution control and observation. I will, in the following, give a short overview over three remarkable methods.
After the adjacent Xerces XML parser has been used to create an internal DOM tree of the WSFL source document, the *initEngine()* method is to be called, searching the DOM tree for constructs constituting the flow. The data is then used to create respective flow elements, such as activities and messages, which are then mapped to their identifier in the engine internal repository.
Each mapping construct found in data links is inserted into an engine internal array, the mapping table, holding source and target message and part. For actually assigning data to message parts an engine method called *setMessagePart()* is used which, after assigning the actual message part, checks the mapping table and, for every occurrence of the respective message part as a mapping source, inserts the newly assigned value to the message part specified as target as well, thus providing for internal data consistency.
For actual flow execution, the engine provides two methods, *spawn()* for running the complete flow and *step()* for step-wise execution, like familiar from debuggers, with both methods operating in a quite similar manner.
When starting flow execution the method first identifies the start activity, determined by finding an activity no control link is pointing to (Notice that the flow's end activity is determined in a similar way, identifying the activity no control link originates at), setting a local variable *current* to that value. Then a loop is entered, which is iterated as long as the end activity has not been completed. Inside the loop, the current activity is executed; then successors are determined using control links originating at the current activity, and depending on their transition condition's evaluation, the actual successor is identified and assigned to *current*, followed by the next loop iteration.

5.1.6 Message Handler Interface

All flow objects provide a method for retrieving their current state, and recursively combined they provide for complete engine repository observation. But there are as well events which have to be handled while flow execution, such as activity and flow start and completion, messages and errors. For this sake a messaging interface is provided, both implemented by the flow engine and the GUI component, the first just printing messages to stdout while the latter enables fine control over messaging properties (e.g. console messages, pop-up dialogs). The actual message handler can be registered with the engine and is subsequently called on specified events.

5.2 Discussion

As the presented implementation was mainly developed for demonstration and proof-of-concept purposes, it naturally has heavy limitations in its functionality.
The first issue to be mentioned is, that the engine is limited to executing sequential flow models, not in a sense of missing alternatives but support for concurrent flows is missing. This leads to various implications, e.g. activities do not need to have join conditions and multi threading was not to be implemented as well. Another result of the limitation to sequential flows in that sense is, that flows must only have one start activity and correspondingly one actual end activity as well (Sequential flows can indeed have multiple end activities, but in actual execution only one of them is reached). That way, the engine knows perfectly where to start and when an end activity is reached, flow execution is stopping.
Another point is, that messages cannot be constructed in WSFL's sophisticated manner, supplying XML schemas for their structure, they are limited to sequences of message parts, where part typing is not supported. Otherwise the need would arise for the engine to support a somewhat more intelligent mapping and message handling mechanism which was esteemed not suitable for the given extent.
Further, service provider's locator construct supports, at this time, only static locators. The last aspect is that the system does not support, at this time, the flow to be provided to the outside as a web service but rather to be controlled and supervised while being executed.
The prototypical implementation was tested successfully and has proven WSFL's ability to dynamically orchestrate web services to complex workflows. Although the research done and test runs have been performed covering rather simple examples, further scalability is, in the author’s opinion, just a matter of sound programming.
The main architecture has proven to be suitable for easily adding additional features, when the need arises for such extra functionality to be implemented; its open and object-oriented approach allows for exchanging or extending parts of the engine structure. In this context, and based on experience gained in implementing and testing the system, the conclusion has been drawn that WSFL can be considered a mature specification in most aspects. It both provides intuitive analysis of workflows and allows for even naive implementation without significant limitations.

6 Open Issues

Web services composition is most surely an emerging technology with a prosperous future, as it is the missing link between the prospect of interoperability and open standards web services are about to bring, and the tried and tested technique of workflow based business process support. Workflows can now be enhanced to cross organizational borders and gain increased dynamics by choosing the participating parties at run-time, thus providing a wide palette of new possibilities to be used by networked, and to-be networked, enterprises. On the other hand the dilemma of web services being restricted to atomic, stateless operations can be solved by integrating them within an orchestrating workflow, providing for the necessary state and logic control context.
This new technique of web services composition is, as presented in part four, most suitable for the development of distributed business processes using the P2P approach, which separates the overall aspects like interface definition and public flow from the concerns of each participating organization. This separation of abstract architecture and interface from the actual implementation of the distributed business process can be perceived like a transformation of the software development process on business applications. A new business process can now be designed in its abstract architecture, defining the participating parties and their common interfaces as a WSFL global model in addition to the public flow model, representing the essence of the overall workflow. The actual implementation of the numerous parts as private flow models can be left to the participants, which have both an interface definition in form of the global model and their public part of the flow model as an abstract guideline for implementation, enabling them to develop a concrete private flow following the extension rules given in [van der Aalst and Weske, 2001]. This separation of concerns paves the way for further modularization in the field of distributed business process design.
One application of resulting workflow distribution apart from interorganizational collaboration is the usage of vertically structured workflows inside enterprises. Common business processes can be modeled by executive designers as public flows, leaving the implementation of the private parts to the respective departments, thus complex processes can be continuously refined as they are successively implemented down the hierarchy. This dynamic structuring of enterprise-internal processes can both boost the enterprise's flexibility regarding such workflows and, through a separation of concerns, provide for the best possible solution, as every part of the overall flow is designed by the respective domain expert.
It is clear that, using the P2P approach in conjunction with web services orchestration, a top-down design can be applied to distributed workflows, featuring continuous refinement in multiple stages. Yet, the question arises, if this technique was not as well suitable for bottom-up development of new complex workflows, composing them from already existing flows. This would allow for one great step ahead, bringing forth the idea of re-usability of workflows in the same way class libraries are used in software engineering. Although great opportunities would arise from such an approach, it is esteemed not feasible, as the maintainers of the workflows used as private flows would have to know about this, because future alteration of theses flows was to follow the given rules and had to keep to legacy interfaces.
Further standardization is one major prerequisite for such re-usable workflows, at least the issue has to be solved, how

distributed business processes are to be provided to users: One obvious part is providing the lifecycle interface for end users of the flow through discovery databases like UDDI strives to establish. But UDDI is currently not intended to provide for flow descriptions, allowing possible private part service providers to plug into an existing public flow or, the other way round, finding an alternative private part service provider from the public flow point of view. This aspects are not all too urgent, as distributed business processes are mostly intended to be long-term partnerships and mutual trust and familiarity of the participants are widely considered necessary prerequisites. But the need for solutions to this problems may arise, once overall market dynamics are increasing and highly dynamic distributed workflows become a reality.

7 References

[Keller et al., 2001] Frank Keller, Peter Tabeling, Rémy Apfelbacher, Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel and Oliver Schmidt: ***Improving Knowledge Transfer at the Architectural Level: Concepts and Notations.*** The 2002 International Conference on Software Engineering Research and Practice, 2002, CSREA Press (to appear). Available online at: http://www.hpi.uni-potsdam.de/source/fachgebiete/modellierung/keller et al 2002-improving knowtrans on archlevel.pdf

[Leyman, 2001] Frank Leymann: ***Web Services Flow Language (WSFL 1.0)*** Draft Standard Publication of the IBM Software Group, available online at: http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[van der Aalst and Weske, 2001] W. M. P. van der Aalst, M. Weske: ***The P2P Approach to Interorganizational Workflows***. Proceedings of the 13th Conference on Advanced Information Systems Engineering (CaiSE'01), pp 140-156, Springer Lecture Notes in Computer Science 2068, Heidelberg, Springer 2001; available online at: http://tmitwww.tm.tue.nl/staff/wvdaalst/Publications/p126.pdf

[fLowRider] C. W. Günther: ***fLowRider*** - Prototypical Implementation of a WSFL flow engine for demonstration and research purposes. Published under the terms of the GNU GPL, sources available online at: http://flowrider.sourceforge.net/

Appendix A

To further describe the scenario flow model given in illustration 5 and provide for easier readability, a larger and more detailed version is given in the following illustration.