ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and
Composable Distributed Components

Service Specification Language

Deliverable Identifier: D6
Delivery Date: 19 September 2003
Classification: Public Circulation
Authors: Ricardo Jiménez-Peris, Marta Patiño-Martínez. Simon Woodman, Santosh Shrivastava, Doug Palmer, Stuart Wheater, Bettina Kemme, Gustavo Alonso.
Document version: 1.0 17 September 2003

Contract Start Date: 1 September 2002
Duration: 36 months
Project coordinator: Universidad Politécnica de Madrid (Spain)
Partners: Universitá di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Universitá degli Studi di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd. (UK)

Project funded by the
European Commission under the
Information Society Technologies
Programme of the 5th Framework
(1998-2002)

CONTENTS

1. Introduction ……………………………………………………………………… 2
2. Web Service Description Language……………………………………………….3
3. Web Service Attributes Related to Transactional Semantics ……………………..5
4. Web Service Specification Related to Performance Properties…………………..15
5. Web Service Attributes Related to Interaction Constraints………………………16

References ………………………………………………………………………..58

1 Introduction

A web service can be either a basic service (BSs) or a composite service (CSs). BSs are provided by a single organization while CSs can be inter-organizational and are made up of BSs and other CSs. The composition of BSs and CSs into a new CS requires specifying the properties of the constituent services in order to predict the properties of the new service. This document deals with a language to specify web services in order to ease the composition of CSs. The specification will cover both functional and non-functional attributes. The description of functional attributes will follow the widely accepted specification for the description of web services, *web service description language* (*WSDL*) [1].

An important part of this document will be devoted to the identification of non-functional aspects of services that will prove helpful to predict the properties of the CSs built out of them. We are interested in enriching the specification of web services in such a way that when they participate in a composition it becomes possible: 1) to find out whether they provide the required functionality to participate in the composition; and 2) the properties of the resulting composition are predictable.

We have identified three families of non-functional attributes that might be of interest for compositions:

- Attributes related to transactional capabilities of web services.
- Attributes dealing with web service performance.
- Attributes regarding sequencing constraints on web service invocations (conversations).

The first and the last set of attributes are static in the sense that their specification does not change after deployment. It is useful to publish static attributes at deployment time. This allows the CS to use the static attributes at composition time to derive the properties of the composition. Performance values, like response times, throughput, and availability, however, change continuously during service execution, that is, they have a dynamic nature. Hence, publishing the values of these dynamic attributes is not as useful as publishing static attributes. However, dynamic attributes are useful at run-time to choose the most appropriate service with respect to its actual performance. For this reason, we will deal differently with the two kinds of attributes. For static attributes, we will define specifications that will be published and then will be used for building compositions. Dynamic attributes, as part of a service, will be associated with an operation. The CS can call this operation whenever it wants to know the current performance values of the web-service. This will help the CS to select a basic service at run-time.
In the following we assume that the web service is provided by a BS and implemented by a *middleware* or *application server*.

2 Web Service Description Language

WSDL has become a de facto standard for the specification of web services. WSDL is both language and platform independent and is defined as an XML grammar. The current version of the language is WSDL 1.1, and WSDL 1.2 is being drafted at this time. In what follows, we will use the terminology as it is in WSDL 1.1.

A WSDL document defines web services as a collection of endpoints operating on messages. The operations and messages are described abstractly to favor reusability, and then, they are linked to a concrete protocol and data format.

Services in WDSL are defined using the following elements:

- types: provide data type definitions used to describe the messages exchanged.
- message: abstract definition of the data exchanged .
- portType: a set of abstract operations and the messages involved.
- binding: specifies a concrete protocol and data format for a portType.
- port: an address for a binding (endpoint).
- service: set of related ports.

Message definitions consist of one or more message *parts*, being each part associated with a type. Port types aggregate messages into collections of operations.

An endpoint can support four transmission primitives:

- One-way. The endpoint receives a message.
- Request-response. The endpoint receives a message and replies with a correlated message.
- Solicit-response. The endpoint sends a message, and receives a reply.
- Notification. The endpoint sends a message.

Two-way operations, request-response and solicit-response, take an input message and if they finish successfully, they return an output message. Notification operations just specify an output message. WSDL currently only defines bindings for one-way and request-response transmission primitives.

Each operation can define fault elements which specify the message format for error messages. Error messages are returned as the output of an operation in case it fails. A fault message can be considered as an exceptional termination that might have an associated value.

Once the abstract part of a service has been defined in terms of port types, operations and messages, its concrete part is defined with the help of bindings. A binding defines the message format and protocol details for the operations and messages of each particular port type. Then, ports define an individual endpoint by associating a concrete address to a binding. Services are used to aggregate a set of ports together.

In the next section we will discuss transactional attributes. We first describe their semantics and then show how they can be specified via WSDL. Section 4 discusses how performance related information can be published. Section 5 discusses in detail attributes regarding sequencing constraints.

3 Web Service Attributes Related to Transactional Semantics

One of the conclusions of the deliverable D5, *Transactional Support*, was that traditional ACID transactions are not applicable in the composition of web services in order to preserve the autonomy of the organizations that provide these services. The main problem is the isolation property. A long lasting transactional composite service can block the resources of the organizations providing the web-services for long periods of time. However, the atomicity property is an essential property in all the advanced transaction models suitable to be used for the composition of web services. This does not mean that individual operations provided by a web service can not be executed as ACID transactions. What is not suitable, in general, is to execute a CS as an ACID distributed transaction.

3.1. Transactional Attributes

The goal of this section is to identify attributes that help to build transactional compositions of web services. These attributes, given the above stated restrictions, will be mainly related to atomicity.
Atomicity guarantees that all or none of the operations of a transaction are executed. This means, that before a transaction commits, all participants must have enough information on stable storage to reproduce the effects of a committed transaction in case of failure (log redo information). Furthermore, each participant must be able to undo already executed operations until it is secure that the transaction will commit. For that purpose undo information must be logged. Such undo information can be physical (containing the before-image of changed objects) or logical (containing the logical reverse operation). Which kind of undo log will be used will depend on the transactional attributes of the web-service.
The transactional-related attributes identified so far are:

- Isolated.
- Atomic.
- Conversational
- Coordinatable
- NonUpdate
- Retriable
- Idempotent
- Compensatable
- Testable

Important issue is how these attributes are implemented. Some of them can be enforced internally by the application server in which the web service is deployed. For instance, an EJB application server implementing the J2EE specification provides atomicity, and all methods of beans deployed in the server can be specified as atomic. Other attributes require help from the web-service provider. For instance, whether an operation is read-only (NonUpdate) can only be determined by the programmer.
The first four attributes directly specify transactional properties. The second five attributes are not directly related to atomicity, but they are useful for the CS level to

build atomic composite services. We would like to foresee that every web-service will provide all attributes.
The attributes listed above are all applicable to the individual operations of a web service. In the following, if it becomes clear from the context, we write “attribute of a web-service” instead of “attribute of an operation of a web-service”. In the following, we will look at each attribute individually. We fill first give a detailed descriptions of its semantics. Then we outline how the property could be implemented. This discussion does not attempt to be exhaustive but only sketches implementation alternatives. Whenever it is interesting, we will also analyze how the different attributes are related to each other.

3.1.1. Isolated Attribute

The **isolated** attribute indicates that a service provides some level of isolation. Well known are the isolation levels for transactions. For instance, *serializability* isolation guarantees that the effect of concurrent execution of transactions is equivalent to a serial execution of them providing the illusion to the multiple users that their transactions have been executed in the system without the interference of other users. Lower levels of isolation have been proposed such as the ANSI isolation levels [13] or Oracle *snapshot isolation* [22]. The ANSI isolation levels, in addition to *serializable*, are *read uncommitted*, *read committed* or *repeatable reads*. The three of them provide increasing levels of isolation all below serializability. The Oracle snapshot isolation lies somewhere between the read committed and serializability isolation levels.

However, isolation levels can also be specified outside of the context of transactions. For instance, *linearizability* [20] guarantees that concurrent invocations to a service are equivalent to a serial execution of them. While serializability provides isolation at the level of a sequence of invocations (i.e., transaction), linerizability provides isolation at the level of an individual invocation.

In any case, the isolation property is provided by the application server hosting the web-service, or by the web-service provider through special means. The client has no control over the transaction/invocation.
For instance, EJB application servers following the J2EE specification, guarantee transactional isolation. The service provider must enclose the whole service code within a transaction and set the isolation level according to the isolated attribute. The application server provides isolation by either relying on the concurrency control of the underlying database systems (and disallowing any concurrent data access within the beans implementing the business logic) or the server implements its own concurrency control strategy (often following on optimistic concurrency control strategy).
Linearizability can be enforced by using short term concurrency control such as mutexes. A service can be made linearizable by using in its code mutexes that protect the access of shared data accessed by either another invocation to the same service or an invocation to a different service.
The isolation attribute is used to specify any isolation properties of a transaction or invocation. We suggest treating atomicity orthogonal to isolation. Some kinds of services can require atomicity independently of isolation properties. For example, it might be acceptable for a given service that two concurrent invocations observe intermediate results of each other (no isolation required), but it must be guaranteed that

each of them is atomic (all-or-nothing-property required). In other cases, isolation might not be a factor. For instance, stateful session beans of a J2EE always belong to a single client, hence, there is never concurrent access.

3.1.2. Atomic Attribute

If a web-service operation provides the **atomic** attribute, then it is guaranteed that the execution of the operation (including all sub-invocations triggered by this operation) fulfills atomicity, i.e., either all or none of the data changes triggered by the operation are successful. That is, the execution starting with the call to the operation and ending with its return builds a single atomic unit. As with isolation, the client has no control over the transaction. It is only informed that the entire operation was successful or failed. How atomicity is implemented is transparent to the client.

As with isolation, the atomic property can either be guaranteed by the application server hosting the web-service or by the web-service provider. The web-service provider can provide atomicity of an operation by, e.g., carefully catching any exception and logically undoing any effect executed so far. Many application servers implement a transaction manager that supports atomicity, e.g., the EJB application servers following the J2EE specification. The service provider simply needs to enclose the code of a service within a transaction. In contrast, with the CORBA Object Transaction Service [23], providing atomicity (recoverable objects) is a responsibility of the programmer of the object.

It is our opinion that the atomic attribute will be essential for many services.

3.1.3. Conversational Attribute

The atomic attribute only provides atomicity within a single web-service call. The transaction terminates before a return is given to the external caller. However, a CS might want to combine several operations of the same web-service to a single transaction. Web-services with such properties contain the **conversational** attribute.
In this case, the BS must provide a `beginTransaction` and `endTransaction` operation. The CS must first call `beginTransaction`, then call a sequence of web-service operations, and then terminate the transaction with `endTransaction`.

Many application servers (e.g., EJB servers) provide atomicity support for such "conversations". In case of a conversational transaction, the transaction manager has to keep track of transaction context across web-service calls. Application servers who support conversational transactions, are usually also able to put a single call into a transaction. Therefore, a "conversational" web-service is usually also "atomic".
In this case, the question arises which of the two attributes comes into effect at runtime. When an operation is executed, should the corresponding transaction commit before return or not? The behavior can be determined at the time of invocation. If the operation is called without a transactional context, a new transaction is created solely for this operation, and the atomic attribute comes into effect. When the operation is called within a transactional context, the system is handling a conversational transaction.

3.1.4. Coordinatable Attribute

In some cases, it might be necessary to coordinate the execution of a set of web services to provide atomicity guarantees across organizations. Such coordinated execution will be based on one of any of the emerging specifications for transactional coordination, such as BTP (Business Transaction Protocol) [17], WS-Transaction/WS-Coordination [18][5][6], or the just disclosed WS-CAF specifications [19]. Services with this coordination capability will include the **coordinatable** attribute. Services with this attribute will exhibit an interface for participating in at least one coordination protocol (e.g., two phase commit protocol) of one of the existing standards and/or specifications. This interface will enable a service to participate in a distributed transaction. We assume that a web service might either, require, allow or forbid the invocation of a web service in a coordination context. By not exhibiting the coordinatable attribute, the web service forbids its invocation within a coordination context. Otherwise, it is allowed. It can be forced by indicating that the invocation within a coordination context is mandatory. Finally, as part of this attribute the standards, family protocols and protocols supported by the web service are indicated.

The ability to participate in an inter-organization transaction is usually provided by a transaction manager that understands one or more web service transaction standards as the one that will be developed in deliverable *D4 Transactional engine*. The transaction manager has to implement the participant part of the coordination protocol. So far, not many EJB application servers provide such feature. It has to be seen whether it is possible to implement an independent transaction manager that is responsible only for the coordination. This manager must cooperate with the transaction manager of the application server in order to control the transactions. It might be necessary to adjust and/or extend the original transaction manager of the application server.

Usually, a web-service that provides the coordinatable attribute will also provide the atomic and conversational attributes. In particular, this will be the case when the coordinatable attribute specifies a coordination protocol that decides about the outcome of a transaction (in most cases a 2-Phase-Commit protocol). In such case, the web-service gives up its right to decide independently whether to commit a transaction. This has two implications. First, the web-service obviously must provide mechanisms to abort and commit the effects of involved operations. This is equivalent to the atomic attribute. Furthermore, it must keep transactional context across system calls since it cannot commit when it returns the result of an operation but must wait for the final decision. This makes the service implicitly conversational.
In general, however, a coordinatable attribute could also specify some other form of coordination that is not related to transactions and/or atomicity. In this case, it is independent of the atomic and conversational attributes.

3.1.5. NonUpdate Attribute

The **NonUpdate attribute** states that a service does not change the application state. In this category, we find two kinds of services: services that only perform read operations, and services that do not have a relevant effect on the server with respect to other service invocations. For instance, a login operation might have an effect on the server, like

saving the username with a timestamp in a log file. However, its execution is not relevant to other clients of the service, since its invocation does not change data that other clients access.

This attribute states a property of a specific operation. As such, this property must probably be specified by the service provider. Only in seldom cases will the application server be able to detect that an operation obeys the NonUpdate rules.

3.1.6. Retriable Attribute

Service invocations can fail due to different circumstances, such as, loss of messages, server overload, server crashes, etc. In the advent of such scenarios, an important attribute of a service is the possibility of retrying the invocation and guaranteeing its successful execution after a finite number of invocations. Such services will be tagged with the **Retriable** attribute. It should be highlighted that a retriable service has to guarantee that it will succeed eventually (i.e., when all failures like message loss, crash, overload etc. are repaired). Therefore, services like a hotel reservation might not be retriable, since there is no guarantee of success. For instance, the hotel might become fully booked. On the other hand, a service that asks for the availability of a room at a hotel might be retriable.

This attribute expresses a semantic property of a service, and as such must be provided by the implementation of the service. Therefore, it is one of the properties to be identified by the programmer of the service.

3.1.7. Idempotent Attribute

If a client can call the same web-service operation twice but the result on the server is as if at most one of the executions has taken place, then the operation should be tagged with the **idempotent** attribute.
Generally, this property is part of the semantics of the service and the programmer should identify it. However, it is possible to provide some support in the application server to guarantee idempotence in a seamless way. The application server can identify invocations in such a way that it can discard duplicates, therefore, enforcing idempotence. This requires support both at the client and server sides. On the client side, the proxy should identify invocations uniquely. That is, if an invocation is resent, it will be tagged with the same identifier as the first submitted invocation. On the server side, duplicates can be filtered based on the invocation identifiers.

This makes clear that idempotence is different from retriable. A purchase order with a unique identifier might be idempotent (that is, it will not be processed twice), but not retriable (the price of the items might change or might not be available any more).

3.1.8. Compensatable Attribute

Service operations that exhibit the **Compensatable** attribute can be undone logically by executing a compensating operation that reverses logically their effect. In this fashion,

atomicity can be enforced by undoing the effect of a previously invoked operation. The difference between compensatable and atomic ones comes from the fact that a compensatable operation must be compensated by the client (by calling the compensating operation) while undoing the effects of an aborting atomic operation is transparent to the client. Atomic operations are automatically aborted by the server implementing the service in case a failure happens.

Whenever an operation of a web service is tagged as compensatable, the service provider must provide a compensating operation to reverse it. The parameters of the compensating service should be linked by the programmer to the (input or output) parameters of the compensatable operation. Only in seldom cases might the application server be able to automatically generate a compensating operation.

3.1.9. Testable Attribute

Web-service operations might be tagged with the **Testable** attribute**.** This attribute indicates that after executing an operation, it is possible to invoke a special operation to find out whether the testable operation executed successfully. This is useful in situations in which the communication can be lost and there is uncertainty about the success or failure of the service invocation.

In some situations, the application server can provide support for this attribute in a similar way as was suggested for the idempotence attribute. The server can tag messages at the client proxy with unique identifiers. If the server executes invocations within a transactional context, then it can also register this identifier in a special table where completed operations are registered. A testing service would be provided so that given the identifier of an operation one can check whether the operation was executed successfully or not. In the absence of this support from the application server, the programmer can always provide a similar functionality within the business logic. There, for each testable operation, a test-operation must be implemented, that, given the identifier of an invocation of the testable operation, determines whether the invocation was successful or not.

3.1.10. Summary

With the first four attributes, the transactional properties are provided to different degrees internally by the web-service. The client can directly take advantage of these transactional properties, and their implementation is transparent to the client.
The second set of attributes, in contrast, does not directly support transactional properties. However, the attributes can help the client, i.e., the CS, to provide atomicity at the CS level by exploiting the special properties of the web-services. For instance, if the CS wants to call a series of operations, such that some of these operations are compensatable, some are retriable and one is atomic, then it can follow the FLEX transaction model [24-26], and first call all compensatable, then the atomic, and then the retriable operations.

One can argue that some of the transactional attributes are not non-functional. For instance, the conversational attribute requires the specification of a compensating

operation. This operation reflects semantics of the application (and must be provided by the application programmer). The compensating operation might not even exactly restore the system to the state before the execution of the original operation (it only logically undoes the effect but does not restore a before-image). Other attributes might also be considered functional since they expose information about the semantics of the operation (e.g., NonUpdate). However, we believe that it is of advantage for CS developers if all attributes that help to enforce transactional attributes are specified and handled in a similar way (whether they are non-functional or have a tendency to be functional).

3.2. Specification and Publication of Transactions Attributes

Transactional and other non-functional attributes of web services need to be defined in an interoperable way and published. So, when an organization uses some of these web services, it can locate these specifications and then, find out whether they are suitable for the targeted task. For interoperability purposes, XML is becoming a de facto standard. Regarding the publication of this specification, two different issues need to be considered: a) where this document is published, so it can be located by a potential user; b) in which document transactional attributes are included.

UDDI repositories [15] are becoming the standard way of publishing web services. A UDDI repository allows publishing and searching web services in a standard way (through well defined APIs). Records in a UDDI repository hold information about the business, as well as technical information such as, bindingTemplates that provide access points to services and tModels that provide links to technical specifications of services. There is a tModel, the wsdlSpec tModel, that points to the WSDL specification of a web service. But it is also possible to include new kinds of tModels that, for instance, could be useful to point to the specification of non-functional attributes.

There are two options for where to specify non-functional attributes. One option is to include them in the WSDL specification of the service by taking advantage of the WSDL extensibility elements. Another possibility consists in using a separate XML document that specifies the non-functional attributes of the operations of a given web-service. The tModel of this web service (residing in the UDDI registry) can point to this XML document. We will adopt the first option since it seems simpler (it does not need to reference operations from a different document). Moreover, the second option does not offer any additional advantage.

In the following, we first discuss the specification of the attributes and then describe how the services can be published in a UDDI repository.

3.2.1. Specifying Transactional Attributes through WSDL Extensible Elements

Extensibility elements are placeholders for elements from other namespaces. They are allowed at particular points of a WSDL document. In particular in WSDL 1.1, the current available specification, the extensibility elements can be placed at the type

definition, at the operation definition in the binding section and at the port definition in the service section.

Since the attributes we are concerned with apply to operations, the only option with the current specification is to include the description of non-functional attributes in the binding section. It seems that it would be more appropriate to locate them in the portType section, since the non-functional attributes do not depend on bindings. Fortunately, the current draft of WSDL (WSDL 1.2) contemplates extensibility of portTypes, filling the missing functionality in the current version of WSDL. In what follows we will assume that portTypes are extensible.

Most of the presented transactional attributes can be specified by using a keyword to tag the corresponding operation. However, some of them need to refer to another operation. For instance, when an operation is defined as compensatable, there must be an associated compensating operation and an indication of which parameters from the operation to be compensated should be passed to the compensating operation. The testable attribute also needs to refer to a test operation. This is necessary since the testable attribute states that the successful completion of an operation can be tested by means of a test operation. In what follows we describe the XML schema for these attributes.

We will adopt the following namespace prefix for describing the different quality of service attributes.

```
xmlns:txn=http://www.adapt.eu.int/wsdl/txn
```

The attributes that do not need to be completed with another operation are modeled with a single keyword:

```
<xsd:simpleType name="SimpleTransAttributeName">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Atomic"/>
    <xsd:enumeration value="NonUpdate"/>
    <xsd:enumeration value="Retriable"/>
    <xsd:enumeration value="Idempotent"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="SimpleTransAttribute">
  <xsd:element name="TransAttributeName"
               type="SimpleTransAttributeName">
</xsd:complexType>
```

The isolated attribute has a companion string that indicates the isolation level provided by the service. The isolation levels we have identified are enumerated in the following XML type:

```
<xsd:simpleType name="IsolationLevel">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Linearizable"/>
    <xsd:enumeration value="ReadUncommitted"/>
    <xsd:enumeration value="ReadCommitted"/>
```

```
    <xsd:enumeration value="RepeatableRead"/>
    <xsd:enumeration value="SnapshotIsolation"/>
    <xsd:enumeration value="Serializable"/>
  </xsd:restriction>
</xsd:simpleType>
```

With this, the definition of the isolated attribute is then as follows:

```
<xsd:complexType name="IsolatedAttribute">
  <xsd:attribute name="mandatory" type="IsolationLevel"
                         default="Serializable">
</xsd:complexType>
```

The coordinatable attribute is a special case since it has attached a list of accepted coordination protocols. The accepted coordination protocols will be referred by a list of three strings (for instance, a string might be a namespace of a protocol):

- One string for the coordination standard (e.g., for WS-Transaction: http://schemas.xmlsoap.org/ws/2002/08/wstx).
- Another string for the protocol familiy (e.g., for business activity: http://schemas.xmlsoap.org/ws/2002/08/2002/wsba).
- And another one for the particular protocol within the family (e.g. businessAgreementWithComplete).

The definition of the coordinatable attribute is as follows:

```
<xsd:complexType name="CoordinatableAttribute">
  <xsd:sequence>
    <xsd:element name="CoordStandard" type="xsd:string"/>
    <xsd:element name="CoordFamily"   type="xsd:string"/>
    <xsd:element name="CoordProtocol" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="mandatory" type="xsd:boolean" default="false">
</xsd:complexType>
```

Furthermore, the compensatable attributes need to associate the tagged operation with another operation whose parameters depend on the parameters of the operation tagged with the attribute. A compensatable operation must specify which operation should be invoked to undo its effect. That is, the invocation of the compensatable operation and its compensating operation must be correlated. The way in which we model this correlation is by associating some of the (input or output) parameters of the compensatable operation to the compensating operation. For instance, to compensate a hotel room booking operation there must be a compensating operation (`CancelRoomBooking`) as well as list of parameters (for instance, a reservation number) that correlate the cancellation with the reservation. This list of parameters will be parameter names of the input and output messages of the operation to be compensated.
Similar holds for the testable attribute if there exists a test-operation for each testable operation.

These two attributes are defined as follows:

```
<xsd:simpleType name="ComplexTransAttributeName">
```

```
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Compensatable"/>
    <xsd:enumeration value="Testable"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="ParameterList">
  <xsd:list itemType="xsd:string">
</xsd:simpleType>

<xsd:complexType name="MsgTransAttribute">
  <xsd:sequence>
     <xsd:element name="MessageName" type="xsd:string">
     <xsd:element name="MessageParameters type="ParameterList">
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ComplexTransAttribute">
  <xsd:sequence>
    <xsd:element name="TransAttributeName"
           type="txn:ComplexTransAttributeName">
    <xsd:element name="AssociatedMsg" type="MsgTransAttribute">
  </xsd:sequence>
</xsd:complexType>
```

Finally the attributes are defined by the `TransAttribute` type.

```
<xsd:simpleType name="TransAttribute">
  <xsd:union memberTypes="SimpleTransAttribute
         CoordinatableAttribute ComplexTransAttribute">
</xsd:simpleType>
```

Each operation of a web service can have a list of attributes defined as:

```
<xsd:simpleType name="TransAttributeList">
  <xsd:list itemType="TransAttribute">
</xsd:simpleType>
```

Examples

The following example shows how an operation, *BuyOrder*, exhibits two simple attributes (Atomic and Idempotent):

```
<portType name="BuyOrderPortType">
  <operation name="BuyOrder">
    <input message=...>
    <output message=...>
    <txn:TransAttributeList>
      <txn:SimpleTransAttribute Atomic />
      <txn:SimpleTransAttribute Idempotent />
    </txn:TransAttributeList>
    ...
  </operation>
</portType>
```

The following example shows a Deposit operation in a bank branch that is a Coordinatable (through the businessAgreement protocol from WS-Transactions), atomic and serializable service that requires to be invoked in a coordination context:

```
<portType name="BankBranch">
  <operation name="Deposit">
    <input message=...>
    <output message=...>
    <txn:TransAttributeList>
      <txn:IsolatedAttribute Serializable />
      <txn:SimpleTransAttribute Atomic />
      <txn:CoordinatableAttribute>
        <txn:CoordStandard>
          “http://schemas.xmlsoap.org/ws/2002/08/wscoor"
        <txn:CoordStandard/>
        <txn:CoordFamily>
          "http://schemas.xmlsoap.org/ws/2002/08/2002/wsba"
        <txn:CoordFamily />
        <txn:CoordProtocol>
          "businessAgreementWithComplete"
        <txn:CoordProtocol/>
      </txn:CoordinatableAttribute>
    </txn:TransAttributeList>
  </operation>
</portType>
```

The hotel room booking operation (*BookRoom*) is defined as atomic, serializable and Compensatable. Let us assume that as part of the output message of that operation a `BookingReferenceNumber` is returned. The *CancelRoomBooking* operation is the compensating operation for *BookRoom,* and `BookingReferenceNumber` is used as input parameter of this operation.

```
<portType name="Hotel">
  <operation name="BookRoom">
    <input message=...>
    <output message=...>
    <txn:TransAttributeList>
      <txn:IsolatedAttribute Serializable />
      <txn:SimpleTransAttribute Atomic />
      <txn:ComplexTransAttribute>
        <txn:TransAttribute Compensatable />
        <txn:AssociatedMsg>
          <txn:MessageName CancelRoomBooking />
          <txn:MessageParams BookingReferenceNumber />
        </txn:AssociatedMsg>
      </txn:ComplexTransAttribute>
    </txn:TransAttributeList>
  </operation>
</portType>
```

3.3. Publishing Transactional Attributes in UDDI Repositories

A UDDI repository allows the publication of web services as XML data models. The data model has four parts:

- the businessEntity describes the information about the company, contacts, business description, etc.
- the businessService includes information about one or more web services.
- the bindingTemplate provides information about how and where a web service can be accessed.
- tModel stands for technical model and provides links to the technical specifications of a service.

One of the recommended uses of the tModel element is precisely to reference to the WSDL description of a web service [16]. In particular, the wsdlSpec tModel is targeted to include the URL to the WSDL specification of the service in its `overviewURL` field. In the ADAPT project we will follow this recommendation. The WSDL document will contain the attributes of the operations of the published web service. Another possibility would have been to include the transactional specification as a separate document and define a new kind of tModel, TxnSpec tModel, to reference it.

Some authors [14] have recently proposed the inclusion of non-functional, e.g., QoS information, as separate elements (blue pages) in the UDDI service description. This approach has the benefit of being able to use QoS attributes as part of a web service search. However, the main shortcoming of the proposed approach is that it is not (yet) standard. Hence, we do not follow this approach in this paper. However, we believe that if such an approach should become standard, then our specification can be easily adjusted to the new format.

4 Web Service Specification Related to Performance Properties

As discussed in Section 1, performance properties have a dynamic nature. If we include performance properties as part of the specification, then we would have to update the specification whenever such properties change. We believe that the current specification format, as well as UDDI implementations, do not support well frequent changes. Even when we change the specification within short time intervals, the published information will never reflect the exact situation at the web-service. Also, CS might cache specifications, and hence, either the cached versions must be invalidated or the CS must check on a regular basis whether their cached versions are still correct.

Hence we suggest that web-services make performance properties and other QoS parameters accessible through special “performance” operations. The CS can call these operations whenever it wants to have information about the current performance of the web-service. Upon invocation, the operations can return very accurate information. Here, we will describe the simplest approach that consists in defining a port type for querying about any performance metric of any port operation:

```
<element name="OptionalOperation" type="xsd:string"
                                  minOccurs="0" maxOccurs="1"/>

<message name="PMInputMessage">
  <part type="xsd:string" name="Metrics"/>
  <part type="xsd:string" name="Port"/>
  <part element="tns:OptionalOperation" name="Operation"/>
</message>

<message name="PMOutputMessage">
  <part type="xsd:float" name="PerformanceMetrics"/>
  <part type="xsd:string" name="Unit"/>
</message>

<portType name="QueryPerformanceMetricPortType">
  <operation name="QueryPerformanceMetric">
    <input message="tns:PMInputMessage"/>
    <output message="tns:PMOutputMessage"/>
  </operation>
</portType>
```

A more involved approach would not only allow to query the performance metrics of a web service, but also to manage it [21].

5 Web Service Attributes Related to Interaction Constraints

5.1. Motivation

As introduced in Section 2, WSDL [1] is a language intended to describe how a client can access a web service, which includes defining the interface of the web service. This interface is defined in terms of the operations that a web service exposes, and each operation is defined in terms of the messages that it consumes and produces. WSDL is able to specify both synchronous and asynchronous operations of web services. WSDL is also able to specify both emitting and consuming sides, although normally only the operations which consume messages are specified.
WSDL is not designed to offer the user of a web service a semantic description of how the service should be used. Sometimes this can be inferred from the names of the operations by humans although not automatically. For example, we intuitively know that we should call the *login* operation before the *order* operation. We also may be able to realise that if the *login* operation fails we should invoke the *register* operation. However, there is no guarantee that the service designer will have used intuitive names for the operations, nor that they will be in a language which the user will understand. Ideally, we would like to formalize the ability to interpret a description of how a service should be used and what operations should be called to achieve a goal or respond to a failure.
We propose augmenting the WSDL description of a service with interaction constraints defining the order in which the operations of a service should/can be invoked. These interaction constraints should be complete, i.e. describe all legal options available to the user of the service based on the previous operations they have called. It should also be possible to validate that a client for a service conforms to the interaction constraints which have been placed upon it. The client could be a simple program or could be a process definition executed by a process manager system. The language developed to

define such interaction constraints should be concise yet flexible to model all desirable situations. It should be aligned to WSDL as far as possible to prevent redefining those aspects of the service already defined and allow the use of standard interfaces already defined in WSDL. Finally, the language should have formal basis to allow us to reason about it and those definitions which are defined in it. We have chosen to base our language around Π-Calculus [2] which is a calculus for describing and analysing concurrent systems. More details on Π-Calculus and the mapping from the interaction constraints to it are provided in Section 5.6.

For the purposes of the ADAPT project, we wish to be able to validate that a CS defined in the Composition Language (D7) conforms to the interaction constraints defined as part of the Service Specification (D6).

5.2. Related Work

At the moment there is a great deal of effort, in many arenas, being expended to address different aspects of interaction specification. These arenas include standard bodies, such as OASIS and WSC, and commercial companies. Four notable, all recent or ongoing, efforts are Business Process Execution Language for Web Service (BPEL4WS), Web Services Choreography Interface (WSCI), Web Services Conversation Language (WSCL) and Web Services Choreography Working Group. In the following section these efforts will be described.

5.2.1. Business Process Execution Language for Web Services

The Business Process Execution Language for Web Service (BPEL4WS) [7] provides a standard for specifying both business process behaviour and business process interactions. BPEL attempts to describe business process interactions using the mutually visible message exchange behaviour of each of the parties involved in the protocol, without revealing their internal behaviour, such descriptions are called *abstract processes*. BPEL4WS provides a language, encoded in XML, for the formal specification of business interaction protocols. The BPEL4WS language contains a rich set of structured programming style constructs for specifying abstract processes, for example, *sequence*, *while*, *pick*, *invoke, assign, wait* and *send*, in addition there are constructs for fault tolerance, message correlation, variables and explicit dependences between constructs. This rich set of constructs means that specifying an abstract process is comparatively complicated and error prone process, and not readably susceptible to automatic analysis.

5.2.2. Web Services Choreography Interface (WSCI)

The Web Service Choreography Interface (WSCI) [8] is an XML-based interface description language that describes the flow of messages exchanged by a web service participating in choreographed interactions with other services. WSCI describes the observable behaviour of a Web service, but does not address the definition and the

implementation of the internal process. WSCI describes behaviour in terms of temporal and logical dependencies among the exchanged messages, sequencing rules, correlations, exception handling, and transactions. WSCI's interface description language, which is based on XML, is used to capture the modeling concepts of: interfaces, activities and choreographs of activities, processes and units of reuse, properties, context, message correlation, exceptions, transactions and compensation activities, and global model. Like BPEL4WS, WSCI's complexity means that it is not readably susceptible to automatic analysis, and there are no signs that WSCI has been widely adopted or updated version will be published.

5.2.3. Web Services Conversation Language (WSCL)

The purpose of Web Service Conversation Language (WSCL) [9] is to provide a standard for specifying business level conversations. WSCL provides an XML schema for specifying business level conversations that take place at a single Web service. The WSCL notion of a *conversation* is a series of message exchanged between a service-consumer and a service-provider. The WSCL specification models a *conversation* as a finite state machine where state changes are triggered by *interactions*. An *interaction* is the exchange of one or two documents between a service-consumer and a service-provider. The WSCL model supports five types of interactions *Send*, *Receive*, *SendReceive*, *ReceiveSend* and *Empty* (the first four of which maps to the WSDL notions of: one-way, notification, send-response and requested-response). WSCL is simple, and analysable, but does have some limitations, such as, only capable of modeling two party conversations and does not support message inspection. There are no signs that WSCL has been widely adopted or updated version will published.

5.2.4. Web Services Choreography Working Group

The Web Services Choreography Working Group [10] is an initiative at by the World Wide Web Coalition (W3C) and was charted in January 2003. The Working Groups is chartered to "… create the definition of a choreography, language(s) for describing a choreography, as well as the rules for composition of, and interaction among, such choreographed Web services." At this time the Working Groups "First Working Draft Specification" is still in preparation.

5.2.5. Summary

There are a number of notable efforts in the area of interaction specification, but each of the efforts described previous present problems if to be used as a part of the ADAPT service specification. BPEL4WS and WSCI are too complex, WSCL is too simple, and the result of the Web Services Choreography Working Group are not yet available. As a result we have constructed our own interaction specification technique.

5.3. Language Overview

The interaction constraints language defines a series of *protocols* involving *participants* using the extensibility elements provided by WSDL to specify the service to which the constraints refer. This means that WSDL service definitions can specify, in addition the ports the service support, the protocols that are supported by the service. This protocol specification within the WSDL will bind the port types of participants in the protocol specification to the port of actual Web service. For clarity, fragments of WSDL for the services are shown with the examples in Section 5.7.

The types of participants in each protocol described by the interaction constraints are defined using the `participantType` element and sub-elements to define the portTypes which that participant supports. The `participantType` and `protocolType` elements are both child elements of the `interactionConstraints` element which is a direct child of the `wsdl:definitions` element.

The interaction constraints themselves are defined within the `protocol` element which contains a name attribute, allowing references to named protocols (or fragments) to be used, enabling re-use of the constraints and aiding modularity. This feature also allows the designer to use recursive structures, for instance to model cases where operations may be called until a particular response is received. Within the protocol, the role which the service itself is playing is defined first, using the `self` element. Following this are references to the other participants in the protocol, using the `participant` element. Finally, the actual protocol is defined in terms of the following constructs:

- Language constructs:
  - Sequence – perform all child elements in sequence with one starting only when the preceding one has completed
  - Choice – perform exactly one of the child elements
  - Parallel – perform all of the child elements in parallel and complete when all parallel executions have completed
  - Multiple – perform the child elements an arbitrary number of times
  - Nothing – do nothing.
- Communication constructs:
  - Send – asynchronous send
  - Receive – asynchronous receive
  - Service – the server side view of a call. There are three elements associated with a Service: ServiceInput, ServiceOutput and ServiceFault. A ServiceInput receives the input to a call. ServiceOutput and ServiceFault correspond to replying to the client with either the output or fault message defined in the WSDL description
  - Invoke – A client side view of a call. InvokeOutput is analogous to sending the call request and InvokeInput/InvokeFault are used to model receiving the result or fault from a call

The language constructs can be nested to an arbitrary level and it is believed that they can model any possible interaction. This is discussed further in Section 5.2. The four communication constructs match those defined in WSDL. `ServiceInput` and

`InvokeOutput` constructs can also have nested language and communication constructs as children to depict the work that may be done to satisfy a request.

If the designer of the service has specified "full WSDL" for their service, i.e. if the service description contains the WSDL describes the messages which will be produced as well as consumed it is possible to define the interaction constraints in terms of that single WSDL document. However, most services are not defined in this manner so it is necessary to provide an alternative method for these services. To achieve this we provide an "invdef" attribute on the participant element. This is intended to denote that "this portType is the inverse of the other one". For instance, the send operation which is not defined in one WSDL document is the inverse of a receive defined in another WSDL document. Whether or not this other document exists is not relevant to the interaction constraints. This simply allows the language to deal with "incomplete but legal WSDL".

It is an issue for the author of the interaction constraints to decide what level of detail they wish to provide. Some may wish to simply model the client and server interaction, preserving the encapsulation offered by the web service. Other designers may wish to expose the sequencing which happens "behind the scenes" in communicating with other services. The latter case offers advantages in scenarios such as the Indirect Response Example given in Section 5.7.5. It allows the client of a service to fully reason about the service that they are using and gives a form of "causality". We wish to be able to offer both options to a service designer and not constrain them to model only simple interaction involving two parties.

Some services are designed in ways that the content of a message is not obvious from its type – one message can convey multiple meanings. For instance, a *LoginResponse* message type used in the xCBL Order Management Use Case [3] can indicate both success and failure. It is thus necessary to be able to inspect the contents of a message and allow different sequences of operations depending on this content. The construct that we use to allow message inspection with is a `content` attribute which is optional on all of the communication constructs. This attribute must contain a Boolean expression and that expression can include XPATH syntax [4]. This attribute acts as a guard on the ensuing elements of the interaction constraints. The scoping of the guard is to the closing tag of the element if the element is non-empty. If the element is empty, the guard is scoped to the closing tag of the parent element. In most cases, this will be a `sequence` element defining the sequence of operations that are allowed following this content. Example usage of the `content` attribute is shown in the Login and Shop example.

When conversations take place, the participants involved could be known before the protocol starts, we will refer to this as having statically bound participants or the participants may be discovered as the protocol progresses, we will refer to this as having dynamically bound participant, this discovery being deduced from the content of messages within the conversation. Naturally, conversation may have a mixture of both statically and dynamically bound participants. Dynamically bound participants is not an uncommon occurrence in more complicated protocols, such as in Web Services Coordination and Web Services Transaction (WS-C and WS-T) [5][6]. In WS-C, an application may be passed a context containing the address of the coordinator to use. We provide a construct for performing such late bindings using optional attributes on

the communication primitives. The `participantBindingName` attribute specifies the participant to be bound from the message. The `participantBindingContent` attribute must specify an element with the message using XPATH. The return type of the XPATH expression should be a node-set and this set should always contain one single text node. Designers of interaction constraints should ensure that the participants which are dynamically bound play no part in the conversation before they are bound to a concrete service.

As described above, the WSDL extensibility mechanisms are used to define the interaction constraints which a service supports. This is a more elegant solution than redefining and extending the WSDL schema which would involve significant changes to tools and could encounter problems when new versions of WSDL are developed. It is hoped that the proposed solution will not encounter such problems as future versions of WSDL should be "backwards compatible".

The specification of interaction constraints by a UDDI tModel [14] is currently under way. If this is possible, it will enable the specifications to be provided either in a repository of services, or by the service description itself.

5.4. Language Structure

This section provides a reference of the structure of the language and the gives fragments of the W3C XML Schema which defines the language. The full schema can be seen in Section 5.8. Figure 1 shows UML Class diagram showing the structure of the schema:

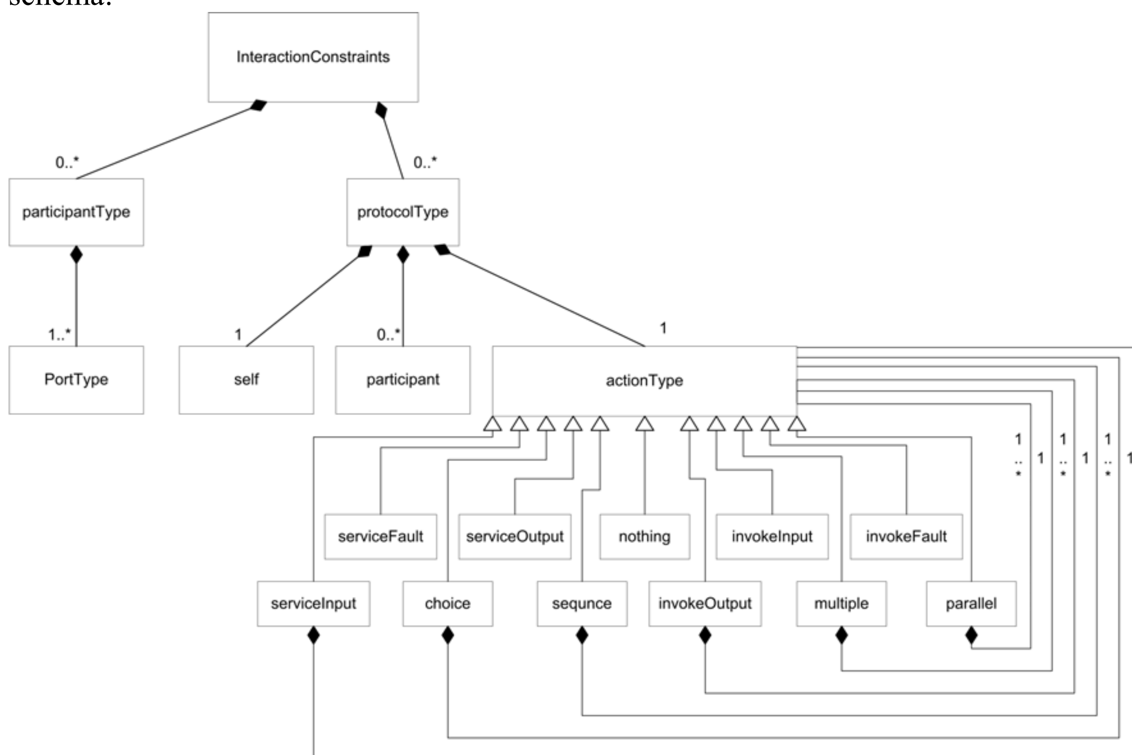


Figure 1, UML class diagram of scheme structure

5.4.1. InteractionConstraints Type

The `InteractionConstraintsType` is the top level type describing the overall constraints placed upon the service.

```
<xsd:complexType name="InteractionConstraintsType">
   <xsd:sequence>
      <xsd:element name="participantType" type="tns:participantTypeType"
      minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="protocolType" type="tns:protocolTypeType" minOccurs="0"
      maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="optional"/>
</xsd:complexType>
```

The `InteractionconstraintsType` has the following sub elements:

- `participantType` – a set of participants in the protocol
- `protocolType` – a set of protocols which this service supports

The `InteractionConstraintsType` has the following attributes:

- `targetNamespace` – the targetNamespace of the document.

Below is an example of the use of the `InteractionConstraintsType` type, it shows that it is used for the "top-level" element of interaction constraints definitions, and contains participant types and protocol types definitions.

```
<interactionConstraints targetNamespace="http://example.org/interaction-constraints1/">
  <participantType …>
    …
  </participantType>
  <protocolType …>
    …
  </protocolType>
</interactionConstraints>
```

5.4.2. ParticipantType Type

The `ParticipantTypeType` is used to describe the participants who take part in the protocol. They are referenced from within the `protocol` element and should have name equivalence.

```
<xsd:complexType name="ParticipantTypeType">
   <xsd:sequence>
      <xsd:element name="portType" type="tns:portTypeType" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:NCName" use="required"/>
</xsd:complexType>
```

The `ParticipantTypeType` has the following sub elements:

- `portType` – a list of the `wsdl:portType` which this participant supports

The `ParticipantTypeType` has the following attributes:

- `name` – the name of the participant

Below is an example of the use of the `ParticipantTypeType` type, it shows that it is used to define a named participant type, and contains port types definitions.

```
<interactionConstraints …>
  <participantType name="participantType1">
    <portType name="portTypeName1" def="servPrefix1:portTypeName1">
    <portType name="portTypeName2" invdef="servPrefix1:portTypeName2">
  </participantType>
  …
</interactionConstraints>
```

5.4.3. PortType Type

The `PortTypeType` structure is used to describe the `wsdl:portType` which the participant implements. These can be directly referenced, or can be “inverse referenced”. The latter case is used to describe situations where the participant contains operations which produce messages but do not consume them. These operations are not usually specified in the wsdl for a service so cannot be referenced. However, they do match the inverse of another operation in another `portType` elsewhere. Thus the service can be thought to implement the inverse `portType` of another service.

```
<xsd:complexType name="PortTypeType">
   <xsd:attribute name="name" type="xsd:NCName" use="required"/>
   <xsd:attribute name="def" type="xsd:QName" use="optional"/>
   <xsd:attribute name="invdef" type="xsd:QName" use="optional"/>
</xsd:complexType>
```

The `PortTypeType` contains no sub elements. The `PortTypeType` contains the following attributes:

- `name` – the name of the portType
- `def` – the qualified definition of the portType
- `invdef` – the qualified inverse definition of the portType

Below is an example of the use of the `PortTypeType` type, it shows that it is used to define a named port type, and references an associated WSDL port types.

```
<interactionConstraints …>
  <participantType name="participantType1">
    <portType name="portTypeName1" def="servPrefix1:portTypeName1">
    <portType name="portTypeName2" invdef="servPrefix2:portTypeName2">
  </participantType>
  …
</interactionConstraints>
```

5.4.4. ProtocolType Type

The `ProtocolTypeType` is used to describe the sequences of operations which the service supports. These can be multiparty and can have an arbitrary level of complexity. It is possible for the interaction constraints to break the encapsulation rule, i.e. expose internal behavour of the service to its clients but this can give a extra level of causality. For instance, in asynchronous systems it may be possible for clients to perform message correlation as they are also aware of the messages that are being sent between services

as well as those being sent to/from themselves. This is shown further in the Indirect Response Example in Section 5.7.3.

```
<xsd:complexType name="ProtocolTypeType">
   <xsd:sequence>
      <xsd:element name="self" type="tns:selfType"/>
      <xsd:element name="participant" type="tns:participantType"
maxOccurs="unbounded"/>
      <xsd:group ref="tns:actionType" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:NCName" use="required"/>
</xsd:complexType>
```

The `ProtocolTypeType` has the following sub elements:

- `self` – defines which participant the service itself is.
- `participant` – a list of the other participants. Each of these must match one of the abstract participants defined in the ParticipantType element.
- `actionType` – description of the protocol itself using the action language constructs described below.

The `ProtocolTypeType` has the following attributes:

- `name` – a unique name for the protocol.

Below is an example of the use of the `ProtocolTypeType` type, it shows that it is used to define a named protocol type, and contains a self, participants and an action definitions.

```
<protocolType name="protocolType1">
  <self type="participantType1"/>
  <participant name="participant1" type="participantType2"/>
  <sequence>
    …
  </sequence>
</protocolType>
```

5.4.5. SelfType

The `SelfType` is used to describe which participants perspective the interaction constraints relate to, i.e. whose interaction constraints these are.

```
<xsd:complexType name="SelfType">
   <xsd:attribute name="type" type="xsd:QName" use="required"/>
</xsd:complexType>
```

The `SelfType` has no sub elements. The `SelfType` has the following attributes:

- `type` – the type of the participant.

Below is an example of the use of the `SelfType` type, it shows that it is used to specify the type participant that self is.

```
<protocolType name="protocolType1">
  <self type="participantType1"/>
  <participant name="participant1" type="participantType2"/>
  <sequence>
    …
```

```
  </sequence>
</protocolType>
```

5.4.6. ParticipantType

The `ParticipantType` element is used to name all of the participants who take part in the interaction. There must be at least one participant listed.

```
<xsd:complexType name="ParticipantType">
   <xsd:attribute name="name" type="xsd:NCName" use="required"/>
   <xsd:attribute name="type" type="xsd:QName" use="required"/>
   <xsd:attribute name="abstract" type="xsd:boolean" use="optional"
default="false"/>
</xsd:complexType>
```

The `ParticipantType` has no sub elements. The `ParticipantType` has the following attributes:

- `name` – a unique name for the participant.
- `type` – the type of participant. This must match one of the participant types described using the participantType element.
- `abstract` – if set to true this allows a participant to be “bound” during the protocol from data in one of the messages sent/received. The default is false.

Below is an example of the use of the `ParticipantType` type, it shows that it is used to define a named participant, and specify its participant type.

```
<protocolType name="protocolType1">
  <self type="participantType1"/>
  <participant name="participant1" type="participantType2"/>
  <sequence>
    …
  </sequence>
</protocolType>
```

5.4.7. ProtocolRefType

The `ProtocolRefType` element allows references to other protocols to appear within a protocol. This aids modularity of design and allows re-use of common protocols. It is hoped that commonly used public protocols such as WS-T will be defined once and then referenced using this element. It is possible to map the participants from the surrounding protocol to participants named in the referenced protocol. This is achieved using the participant sub element and if the participant sub element is not present it is assumed that the participants will be bound during the referenced protocol.

```
<xsd:complexType name="ProtocolRefType">
      <xsd:sequence>
         <xsd:element name="participant" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
               <xsd:attribute name="name" type="xsd:NCName"/>
               <xsd:attribute name="ref" type="xsd:QName"/>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="ref" type="xsd:QName" use="required"/>
   </xsd:complexType>
```

The `ProtocolRefType` has the following sub elements:

- participant – the mapping for a participant. The name attribute must match a named participant in the referencing protocol. The ref attribute must match one of the named participants in the referenced protocol.

The `ProtocolRefType` has the following attributes:

- `ref` – the qualified name of the protocol being reference. This must match a named protocol.

Below is an example of the use of the `ProtocolRefType` type, it shows that it is used to define a sub protocol, and references that protocol's name.

```
<sequence>
  <send …/>
  <protocolRef ref="servPrefix1:protocolName1"/>
  <send …/>
</sequence>
```

5.4.8. SequenceType

The `SequenceType` element is used within the protocol element to denote that certain operations must happen in sequence, i.e. one operation cannot happen until the preceding one has completed. A sequence should have at least two actions which are being sequenced. A `sequence` element should not appear directly below another `sequence`. This is the same as just using the outer `sequence` element. Equally a `sequence` element should not directly follow another `sequence` element. This case is equivalent to concatenating the two `sequence`s.

The `SequenceType` maps to the sequence (.) operator in π-calculus.

```
<xsd:complexType name="SequenceType">
   <xsd:group ref="tns:ActionType" minOccurs="2" maxOccurs="unbounded"/>
</xsd:complexType>
```

The `SequenceType` may have any of the `ActionType` elements as sub elements. The `ActionType` elements are described in Section 5.4.23. The `SequenceType` has no attributes.

Below is an example of the use of the `SequenceType` type, it shows that it is used to define a `sequence` tag.

```
<sequence>
  <send …/>
  <receive …/>
  <send …/>
  <receive …/>
  <send …/>
  <receive …/>
</sequence>
```

5.4.9. ChoiceType

The `ChoiceType` is used to describe situations where there is a choice of actions which can occur in the protocol. For instance, either send a *LoginSuccess* message or send a

LoginFailure message. Often the choice construct will be combined with a `sequence` construct to allow the protocol to proceed down one of two different “paths”. A choice element must have at least two children elements and exactly one of these will be selected.

The `ChoiceType` maps to the choice (+) operator in π-calculus.

```
<xsd:complexType name="ChoiceType">
   <xsd:group ref="tns:ActionType" minOccurs="2" maxOccurs="unbounded"/>
</xsd:complexType>
```

The `ChoiceType` may have any of the `ActionType` elements as sub elements. The `ActionType` elements are described in Section 5.4.23. The `ChoiceType` has no attributes.

Below is an example of the use of the `ChoiceType` type, it shows that it is used to define a `choice` tag.

```
<choice>
  <receive …/>
  <receive …/>
  <receive …/>
</choice>
```

5.4.10. ParallelType

The `ParallelType` element is used to depict actions or sets of actions (using the sequence element) which can occur in any order. All of the actions must occur but the order can be arbitrary.

The `parallelType` maps to the parallel Composition (|) operator in π-calculus.

```
<xsd:complexType name="ParallelType">
   <xsd:group ref="tns:ActionType" minOccurs="2" maxOccurs="unbounded"/>
</xsd:complexType>
```

The `ParallelType` may have any of the `ActionType` elements as sub elements. The `ActionType` elements are described in Section 5.4.23. The `ParallelType` has no attributes.

Below is an example of the use of the `ParallelType` type, it shows that it is used to define a `parallel` tag.

```
<parallel>
  <send …/>
  <send …/>
  <send …/>
</parallel>
```

5.4.11. MultipleType

The `MultipleType` element describes actions which can happen multiple times. The number of times that they can happen is not known until run-time. The `MultipleType`

element must have exactly one child element to denote the structure which will be executed multiple times although this could be a SequenceType so a sequence of operations can happen multiple times.

The MultipleType maps to the replication (!) operator in π-calculus.

```
<xsd:complexType name="MultipleType">
   <xsd:group ref="tns:ActionType"/>
</xsd:complexType>
```

The MultipleType may have any of the ActionType elements as sub elements. The ActionType elements are described in Section 5.4.23. The MultipleType has no attributes.

Below is an example of the use of the MultipleType type, it shows that it is used to define a multiple tag.

```
<multiple>
  <sequence>
    <send …/>
    <receive …/>
  </sequence>
</multiple>
```

5.4.12. NothingType

The NothingType is used to allow an optional step in the interaction constraints. When combined with a ChoiceType element it allows a protocol to continue whether or not an action or set of action occurs. The NothingType element must be empty.

```
<xsd:simpleType name="NothingType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value=""/>
   </xsd:restriction>
</xsd:simpleType>
```

The NothingType has no sub elements. The NothingType has no attributes.

Below is an example of the use of the NothingType type, it shows that it is used to define a nothing tag.

```
<nothing/>
```

5.4.13. SendType

The SendType models an operation which performs an asynchronous sending of a message. This equates to a wsdl:operation which has only an output message. This type of communication can also be referred to as a notification.

The author of the interaction constraints must qualify which operation the send is referring to using the InteractionAttributes described in Section 5.4.21. As

described above, the portType used may be defined explicitly but it is more likely in this situation that it is defined as the inverse of another portType defined elsewhere.

It is possible to refer to the contents of the message that is being sent using the XpathAttribute group. These allow the binding of a participant from the message or give different subsequent operations based on the contents of the message. These options are explained in detail in section 5.4.22.

```
<xsd:complexType name="SendType">
   <xsd:attributeGroup ref="tns:InteractionAttributes"/>
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The SendType has no sub elements. The SendType has attributes described fully in Section 5.4.21 and 5.4.22.

Below is an example of the use of the SendType type, it shows that it is used to define a send tag.

```
<send operation="oper1" portType="port1" participant="participant2"
      participantPortType="port2"/>
```

The send tag example below contains a content attribute that indicates the required content of the sent message.

```
<send operation="oper1" portType="port1" participant="participant2"
      participantPortType="port2" content="boolean(//success)"/>
```

5.4.14. ReceiveType

The ReceiveType element models an asynchronous receipt of a message. The author of the constraints may use the message inspection attributes (5.4.22) with this element and must use the InteractionAttributes to describe the receiving operation as described in Section 5.4.21

```
<xsd:complexType name="ReceiveType">
   <xsd:attributeGroup ref="tns:InteractionAttributes"/>
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The ReceiveType has no sub elements. The ReceiveType has attributes described fully in Section 5.4.21 and 5.4.22.

Below is an example of the use of the ReceiveType type, it shows that it is used to define a receive tag.

```
<receive operation="oper1" portType="port1" participant="participant2"
         participantPortType="port2"/>
```

The receive tag example below contains a content attribute that indicates the required content of the received message.

```
<receive operation="oper1" portType="port1" participant="participant2"
         participantPortType="port2" content="boolean(//success)"/>
```

5.4.15. InvokeOutputType

The `InvokeOutputType` action is used to describe the sending of a message which provides input to another service, implemented as a call. It is used with the `InvokeInput` and `InvokeFault` to model a call to a synchronous service.

The `InvokeOutputType` can have an arbitrary nesting structure of actions underneath it. These should include at least one `InvokeInput` or `InvokeFault` to model the return from the call. It may be possible to have a number of "return points" where the call can return to (possibly depending on the message contents). These are modeled with multiple `InvokeInput` and `InvokeFault` elements.

```
<xsd:complexType name="InvokeOutputType">
   <xsd:group ref="tns:ActionType" maxOccurs="unbounded"/>
   <xsd:attributeGroup ref="tns:InteractionAttributes"/>
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The `InvokeOutputType` element may have any of the `ActionType` elements as sub elements. The `ActionType` elements are described in Section 5.4.23. The `InvokeOutputType` has attributes described in Section 5.4.21 and 5.4.22.

Below is an example of the use of the `InvokeOutputType` type, it shows that it is used to define an operation invocation, and contains an action definitions.

```
<invokeOutput operation="oper1" portType="port1" participant="participant2"
              participantPortType="port2">
  <choice>
    <invokeInput/>
    <invokeFault name="faultName"/>
  </choice>
</invokeOutput>
```

5.4.16. InvokeInputType

The `InvokeInputType` models the normal return from a synchronous invocation of a web service, i.e. the receipt of a `wsdl:output` message. This element should always be a child (although not necessarily a direct child) of the `InvokeOutputType`.

```
<xsd:complexType name="InvokeInputType">
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The `InvokeInputType` has no sub elements. The `InvokeInputType` can have any of the `XpathAttributes` described in Section 5.4.22.

Below is an example of the use of the `InvokeInputType` type, it shows that it is used to define an `invokeInput` tag.

```
<invokeOutput operation="oper1" portType="port1" participant="participant2"
              participantPortType="port2">
  <choice>
    <invokeInput/>
```

```
    <invokeFault name="faultName"/>
  </choice>
</invokeOutput>
```

5.4.17. InvokeFaultType

The `InvokeFaultType` models the receipt of a fault message from a synchronous invocation of a web service. It should always be the child element of a `InvokeOutputType` (although not necessarily the direct child).

```
<xsd:complexType name="InvokeFaultType">
   <xsd:attribute name="name" type="xsd:NCName" use="required"/>
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The `InvokeFaultType` has no sub elements. The `InvokeFaultType` can have any of the `XpathAttributes` described in Section 5.4.22

Below is an example of the use of the `InvokeFaultType` type, it shows that it is used to define an `invokeFault` tag.

```
<invokeOutput operation="oper1" portType="port1" participant="participant2"
              participantPortType="port2">
  <choice>
    <invokeInput/>
    <invokeFault name="faultName"/>
  </choice>
</invokeOutput>
```

5.4.18. ServiceInputType

The `ServiceInputType` element describes the input to a call which is exposed by the service providing the interaction constraints. It is combined with the `ServiceOutputType` and `ServiceFaultType` elements to model a full synchronous call made to the service. `ServiceOutputType` and `ServiceFaultType` should always appear "underneath" the `ServiceInputType` to describe when the response is returned to the client.

```
<xsd:complexType name="ServiceInputType">
   <xsd:group ref="tns:ActionType"/>
   <xsd:attributeGroup ref="tns:InteractionAttributes"/>
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The `ServiceInputType` element may have any of the `ActionType` elements as sub elements. The `ActionType` elements are described in Section 5.4.23. The `ServiceInputType` has attributes described in Sections 5.4.21 and 5.4.22.

Below is an example of the use of the `ServiceInputType` type, it shows that it is used to define a service invocation, and contains an action definitions.

```
<serviceInput operation="oper1" portType="port1" participant="participant2"
              participantPortType="port2">
```

```
  <choice>
    <serviceOutput/>
    <serviceFault name="faultName"/>
  </choice>
</serviceInput>
```

5.4.19. ServiceOutputType

The `ServiceOutputType` describes the response sent to a client who invoked a call on the service described by the interaction constraints. It should always appear as a child element of the `ServiceInputType` element.

```
<xsd:complexType name="ServiceOutputType">
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The `ServiceOutputType` has no sub elements. The `ServiceOutputType` has attributes described in Section 5.4.22.

Below is an example of the use of the `ServiceOutputType` type, it shows that it is used to define a `serviceOutput` tag.

```
<serviceInput operation="oper1" portType="port1" participant="participant2"
              participantPortType="port2">
  <choice>
    <serviceOutput/>
    <serviceFault name="faultName"/>
  </choice>
</serviceInput>
```

5.4.20. ServiceFaultType

The `ServiceFaultType` element models returning a `wsdl:fault` message to a client in response to a synchronous invocation of a web service exposed by the service described by the interaction constraints.

The `ServiceFaultType` element should always appear as a child of the `ServiceInputType` element as it is not possible to return a fault to a call which has not been made/has completed.

```
<xsd:complexType name="ServiceFaultType">
   <xsd:attribute name="name" type="xsd:NCName" use="required"/>
   <xsd:attributeGroup ref="tns:XpathAttributes"/>
</xsd:complexType>
```

The `ServiceFaultType` has no sub elements. The `ServiceFaultType` has attributes described in Sections 5.4.21 and 5.4.22.

Below is an example of the use of the `ServiceFaultType` type, it shows that it is used to define a `serviceFault` tag.

```
<serviceInput operation="oper1" portType="port1" participant="participant2"
              participantPortType="port2">
  <choice>
    <serviceOutput/>
```

```
    <serviceFault name="faultName"/>
  </choice>
</serviceInput>
```

5.4.21. InteractionAttributes

The `InteractionAttributes` are used on the communication primitives described in Sections 5.4.13 – 5.4.20. They are used to define the operation which is being referred to by the action in the interaction constraints. The attributes consist of:

- `portType` – the portType that of the service being referred to.
- `operation` – the operation of the portType specified above.
- `participant` – the participant which is involved in the action. This must match one of the `participant`s named at the start of the protocol.
- `participantPortType` – the portType of the participant.
- `participantOperation` – the operation of the portType on the participant involved. The attribute is optional and only necessary if there are multiple operations on the participant with the same signature.

The above attributes allow the action to refer to exactly one operation on both of the parties involved (self and the participant). All the values of the attributes must be non-colonised names. [12]

```
<xsd:attributeGroup name="InteractionAttributes">
   <xsd:attribute name="operation" type="xsd:NCName" use="required"/>
   <xsd:attribute name="portType" type="xsd:NCName" use="required"/>
   <xsd:attribute name="participant" type="xsd:NCName" use="required"/>
   <xsd:attribute name="participantPortType" type="xsd:NCName" use="required"/>
   <xsd:attribute name="participantOperation" type="xsd:NCName" use="optional"/>
</xsd:attributeGroup>
```

5.4.22. XpathAttributes

The `XpathAttributes` allow the author to inspect the contents of a message. This can be used for two purposes:

- Dynamic Binding of participants: Situations exist when the details of some of the participants of a protocol are not known until the protocol is under way, for instance WS-C [6]. The `participantBindingName` and `participantBindingContent` attributes allow participants to be bound during the protocol. The former attribute specifies the name of the participant to be bound. The latter must contain an XPATH expression to identify the fragment of the message which contains the binding details of the participant. This expression should return a node-set which contains a single node.
- Different protocols based on message content: It is desirable to be able to specify different options for the protocol depending on the contents of a message. For instance, xCBL specifies an order management scenario where a *LoginResponse* message can indicate both success or failure of the *Login* operation. Clearly we wish to be able to distinguish between the two cases and only allow an order to be placed if the login has been successful. The content attribute specifies an XPATH expression which must evaluate to a Boolean value. This attribute acts as a Boolean guard on the remainder of the protocol.

For instance, if used on a receiving action within a sequence and the expression evaluates to:

- `true`: the message was received and the protocol continues.
- `false`: the message was received but the protocol does not continue down this “track”.

```
<xsd:attributeGroup name="XpathAttributes">
   <xsd:attribute name="content" type="xsd:string" use="optional"/>
   <xsd:attribute name="participantBindingName" type="xsd:NCName" use="optional"/>
   <xsd:attribute name="participantBindingContent" type="xsd:string"
use="optional"/>
</xsd:attributeGroup>
```

5.4.23. ActionType

The `ActionType` element group is the group of elements which can be nested below the `protocolType`.

```
<xsd:group name="ActionType">
   <xsd:choice>
      <xsd:element name="sequence" type="tns:SequenceType"/>
      <xsd:element name="choice" type="tns:ChoiceType"/>
      <xsd:element name="parallel" type="tns:ParallelType"/>
      <xsd:element name="multiple" type="tns:MultipleType"/>
      <xsd:element name="nothing" type="tns:NothingType"/>
      <xsd:element name="send" type="tns:SendType"/>
      <xsd:element name="receive" type="tns:ReceiveType"/>
      <xsd:element name="invokeOutput" type="tns:InvokeOutputType"/>
      <xsd:element name="invokeInput" type="tns:InvokeInputType"/>
      <xsd:element name="invokeFault" type="tns:InvokeFaultType"/>
      <xsd:element name="serviceInput" type="tns:ServiceInputType"/>
      <xsd:element name="serviceOutput" type="tns:ServiceOutputType"/>
      <xsd:element name="serviceFault" type="tns:ServiceFaultType"/>
      <xsd:element name="protocol" type="tns:ProtocolRefType"/>
   </xsd:choice>
</xsd:group>
```

The `ActionType` sub elements are described in Sections 5.4.8 to 5.4.20. The `ActionType` attributes are described in Sections 5.4.8 to 5.4.20.

5.5. Language Use Reference

This section describes how the language should be used within the WSDL description of a service to define the interaction constraints. The schema for this can be seen in Section 5.8 and an example usage in Section 5.7.1

5.5.1 ProtocolType Type

This element is used to describe the interaction constraints which the service supports. It is the top level element as far as the interaction constraints are concerned and must be a direct child of the wsdl:definitions element in the WSDL document.

```
<xsd:complexType name="ProtocolType">
  <xsd:sequence>
     <xsd:element name="self" type="tns:SelfType"/>
     <xsd:element name="participant" type="tns:ParticipantType" minOccurs="0"
                  maxOccurs="unbounded"/>
```

```
  </xsd:sequence>
  <xsd:attribute name="type" type="xsd:QName"/>
</xsd:complexType>
```

The protocol element should have a type attribute which references a named protocolType. This should be either defined in or imported into the WSDL document and is used to describe the concrete interaction constraints.

Below is an example of the use of the `ProtocolType` type, it shows that it is used to define a named protocol, and contains a self and participants definitions.

```
<wsdl:service …>
  <wsdl:port …>
    …
  </wsdl:port>
  <protocol type="protocolType1">
    <self>
      <portType name="portName1" port="tns:port1"/>
    </self>
    <participant name="participant1" type="participantType2">
      <portType name="portName2" port="client:port2"/>
      <portType name="portName3" port="client:port3"/>
    </participant>
  </protocol>
</wsdl:service>
```

5.5.2 SelfType Type

The `self` element must appear as the first child element of the `protocol` element. It describes the role that this service plays in the interaction and which portTypes this service supports. Only one self element should appear as the service can only play one role per protocol. The self element should be qualified with at least one sub element to define the portType which the service supports for the protocol.

```
<xsd:complexType name="SelfType">
  <xsd:sequence>
    <xsd:element name="portType" type="tns:PortTypeType" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Below is an example of the use of the `SelfType` type, it shows that it is used to specify define a self, and contains port type definitions.

```
<wsdl:service …>
  <wsdl:port …>
    …
  </wsdl:port>
  <protocol type="protocolType1">
    <self>
      <portType name="portName1" port="tns:port1"/>
    </self>
    <participant name="participant1" type="participantType2">
      <portType name="portName2" port="client:port2"/>
      <portType name="portName3" port="client:port3"/>
    </participant>
  </protocol>
</wsdl:service>
```

5.5.3 ParticipantType Type

The `participant` element must appear directly after the `self` element. It is used to describe the other parties which take part in the interaction constraints. For instance, if the interaction constraints refer to a classical RPC, from a "server" aspect, the only other participant would be a "client". It is possible to specify any number of other participants depending on the complexity of the protocol. Each participant must be qualified with the portTypes that they support, using the portType sub element and the name and port attributes associated with this.

```
<xsd:complexType name="ParticipantType">
  <xsd:sequence>
    <xsd:element name="participant" type="tns:PortTypeType" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName"/>
</xsd:complexType>
```

Below is an example of the use of the `ParticipantType` type, it shows that it is used to specify define a participant, and contains port type definitions.

```
<wsdl:service …>
  <wsdl:port …>
    …
  </wsdl:port>
  <protocol type="protocolType1">
    <self>
      <portType name="portName1" port="tns:port1"/>
    </self>
    <participant name="participant1" type="participantType2">
      <portType name="portName2" port="client:port2"/>
      <portType name="portName3" port="client:port3"/>
    </participant>
  </protocol>
</wsdl:service>
```

5.5.4 PortTypeType Type

The portType element must appear as a direct child of the self element or the participant element. It is used to define the portTypes that the service and participants support.

```
<xsd:complexType name="PortTypeType">
  <xsd:attribute name="name" type="xsd:NCName"/>
  <xsd:attribute name="port" type="xsd:QName"/>
</xsd:complexType>
```

Below is an example of the use of the `PortTypeType` type, it shows that it is used to define a named port type, and references WSDL port specification.

```
<wsdl:service …>
  <wsdl:port …>
    …
  </wsdl:port>
  <protocol type="protocolType1">
    <self>
      <portType name="portName1" port="tns:port1"/>
    </self>
    <participant name="participant1" type="participantType2">
      <portType name="portName2" port="client:port2"/>
      <portType name="portName3" port="client:port3"/>
    </participant>
  </protocol>
</wsdl:service>
```

5.6. Π-Calculus Mapping

To ensure that our interaction constraints language is both complete and precise we have based it on π-calculus, a basic formalism designed for the validation of communicating systems. In [2] Milner proves that communicating systems can be modeled with a relatively small set of language primitives:

- sequence – P.Q, process P followed by process Q.
- summation (or choice) – P+Q, either process P or process Q.
- parallel composition – P|Q, both P and Q in parallel.
- replication - !P, P many times in parallel.
- receive – x(a), receive a message called a on channel x.
- send – x<a>, send a message a on channel x.

The first four primitives describe how to construct processes and the final two describe communication between processes. In our language we use the construction primitives directly and provide several different send and receive primitives for alignment with WSDL. We regard each individual web service as a process in the sequence, larger processes can be created with the language primitives. The language to π-calculus mappings are shown below:

Language construct	π-calculus mapping
<sequence>	sequence
<choice>	summation
<parallel>	parallel
<multiple>	replication
<receive>	receive
<send>	send
<serviceInput>	receive
<serviceOutput>	send
<serviceFault>	send
<invokeOutput>	send
<invokeInput>	receive
<invokeFault>	receive

Communication channels are defined by the participant attributes of the various communication constructs. `send`, `receive`, `serviceInput` and `invokeOutput` all define their associated participants directly. `serviceOutput` and `serviceFault` use the participant attribute of the associated `serviceInput` tag. Similarly `invokeInput` and `invokeFault` use the participant attribute from the associated `invokeOutput` tag.

The π-calculus mapping described above allows the system developer to verify that they are using a service as the service designer intended. To achieve this, the developer needs to produce a π-calculus representation of their system which it is possible to validate against the π-calculus representation of the interaction constraints. An alternative to

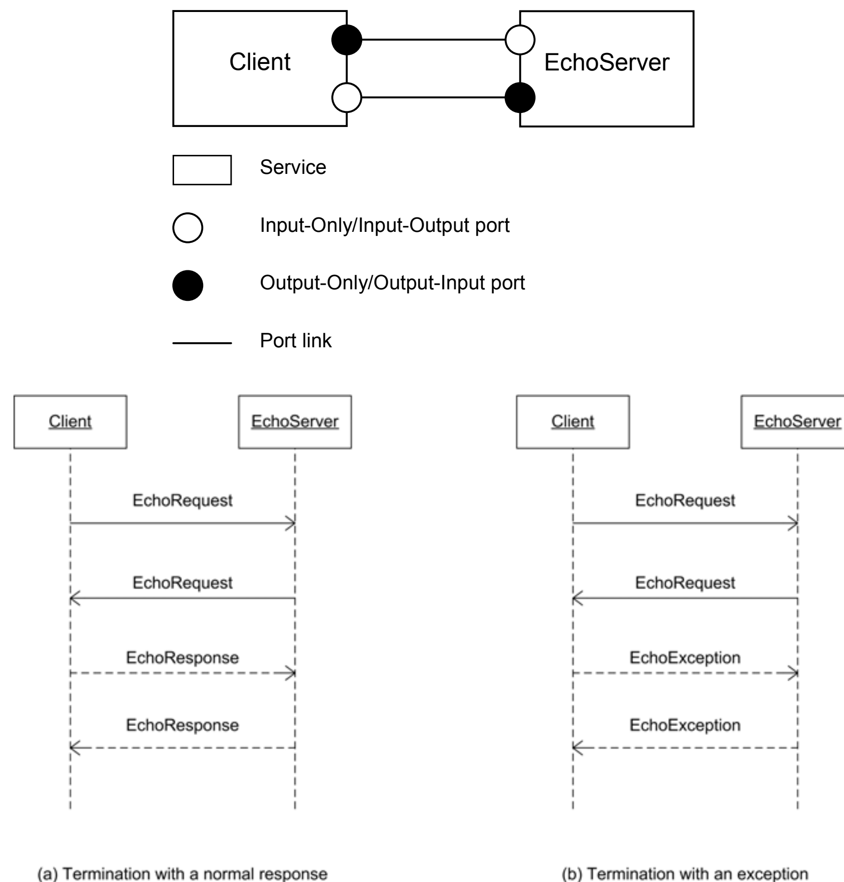
developing their own π-calculus representation could be to use a language which has an inherent mapping to π-calculus such as the one described in D7 [28]. Tools could be used to verify that the π-calculus representation of a system validates against the representation of π-calculus for the sequencing constraints, thus proving that the system utilizes the service correctly. We are currently investigating the Mobility Workbench [27] being developed by Edinburgh University for this purpose.

5.7. Examples

In this section some scenarios involving specification of services interaction constraints will be presented, these scenarios vary from real world protocols to cases which prove difficult to model in other technologies.

5.7.1. Callback Example

In this scenario, the *Client* makes a synchronous invocation of the *echo* operation of the *EchoServer*, which in turn calls the *echo* operation of the *Client*, when the invocation on the *Client* completed the invocation on *EchoServer* completes.



The port types involved in this scenario are:

```
<wsdl:portType name="Echoer">
  <wsdl:operation name="echo" parameterOrder="text">
```

```
    <wsdl:input  message="tns:EchoRequest"/>
    <wsdl:output message="tns:EchoResponse"/>
    <wsdl:fault  name="EchoException" message="tns:EchoException"/>
  </wsdl:operation>
</wsdl:portType>
```

This scenario contains one type of participant:

```
<participantType name="EchoService">
  <portType name="Echoee" invdef="tns:Echoer"/>
  <portType name="Echoer" def="tns:Echoer"/>
</participantType>
```

The protocol specification from the *EchoServer*'s perspective, could look something like:

```
<protocolType name="EchoWithCallback">
  <self type="tns:EchoService"/>
    <participant name="EchoClient" type="tns:EchoService"/>
  <serviceInput operation="echo" portType="Echoer" participant="Client"
participantPortType="Echoee">
      <sequence>
        <invokeOutput operation="echo" portType="Echoee" participant="Client"
participantPortType="Echoer">
          <choice>
            <invokeInput/>
            <invokeFault name="EchoException"/>
        </choice>
      </invokeOutput>
      <choice>
        <serviceOutput/>
        <serviceFault name="EchoException"/>
      </choice>
    </sequence>
  </serviceInput>
</protocolType>
```

The protocol specification from the *Client*'s perspective, could look something like:

```
<protocolType name="CallWithCallback">
  <self type="tns:EchoService"/>
  <participant name="Echoer" type="tns:EchoService"/>
  <invokeOutput operation="echo" portType="Echoee" participant="Server"
participantPortType="Echoer">
    <sequence>
      <serviceInput operation="echo" portType="Echoer" participant="Echoer"
participantPortType="Echoee">
        <choice>
          <serviceOutput/>
          <serviceFault name="EchoException"/>
        </choice>
      </serviceInput>
      <choice>
        <invokeInput/>
        <invokeFault name="EchoException"/>
      </choice>
    </sequence>
  </invokeOutput>
</protocolType>
```

The service specification for the *EchoServer*

```
<wsdl:service name="EchoServer">
  <wsdl:port name="Echoer" binding="tns:EchoerSoapBinding">
    . . .
  </wsdl:port>
  <wsdl:port name="Echoee" binding="tns:EchoeeSoapBinding">
    . . .
  </wsdl:port>
  <protocol type="echo:EchoWithCallback">
    <self>
      <portType name="Echoer" port="tns:Echoer"/>
      <portType name="Echoee" port="tns:Echoee"/>
    </self>
    <participant name="EchoClient">
```

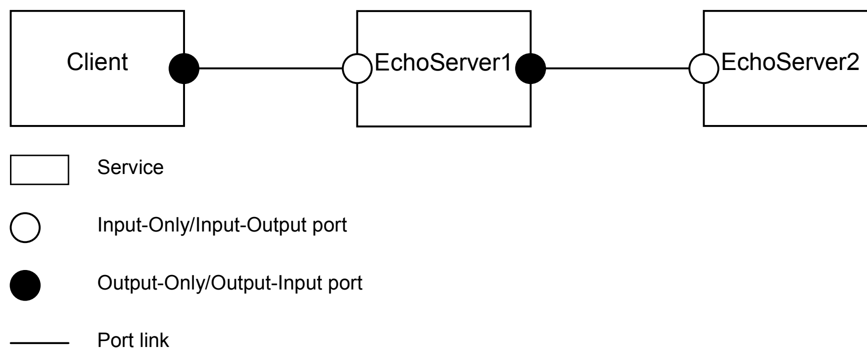
```
      <portType name="Echoer" port="client:Echoer"/>
      <portType name="Echoee" port="client:Echoee"/>
    </participant>
  </protocol>
</wsdl:service>
```

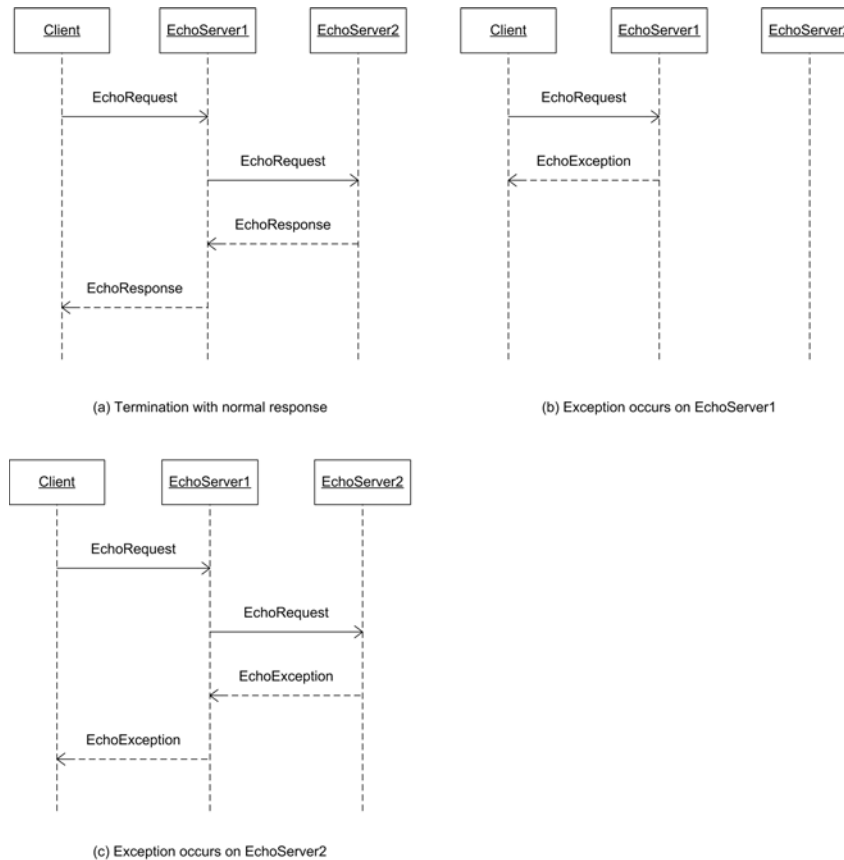
Π-Calculus equivalent

```
CE(EchoReq) . EC<EchoReq> . (CE(EchoRes) + CE(EchoException)) .
EC<EchoRes> + EC<EchoException>)
```

5.7.2. Forwarder Example

In this scenario, the *Client* makes a synchronous invocation of the *echo* operation of the *EchoServer1*, which calls the *echo* operation of the *EchoServer2*, when the invocation on *EchoServer2* is completed the invocation on *EchoServer1* completes. If at any stage an invocation generates a fault, the enclosing invocation, if any, should produce a fault.





(a) Termination with normal response

(b) Exception occurs on EchoServer1

(c) Exception occurs on EchoServer2

The port types involved in this scenario are:

```
<wsdl:portType name="Echoer">
  <wsdl:operation name="echo" parameterOrder="text">
    <wsdl:input  message="tns:EchoRequest"/>
    <wsdl:output message="tns:EchoResponse"/>
    <wsdl:fault  name="EchoException" message="tns:EchoException"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="Echoee">
  <wsdl:operation name="echo" parameterOrder="text">
    <wsdl:output message="tns:EchoResponse"/>
    <wsdl:input  message="tns:EchoRequest"/>
    <wsdl:fault  name="EchoException" message="tns:EchoException"/>
  </wsdl:operation>
</wsdl:portType>
```

This scenario contains three types of participants:

```
<participantType name="EchoClient">
  <portType name="Echoee" def="tns:Echoee"/>
</participantType>

<participantType name="EchoServer">
  <portType name="Echoer" def="tns:Echoer"/>
</participantType>

<participantType name="EchoForwarder">
  <portType name="Echoee" def="tns:Echoee"/>
  <portType name="Echoer" def="tns:Echoer"/>
</participantType>
```

Or, if the port type Echoee was not defined (uses invdef):

```
<participantType name="EchoClient">
  <portType name="Echoee" invdef="tns:Echoer"/>
</participantType>

<participantType name="EchoServer">
  <portType name="Echoer" def="tns:Echoer"/>
</participantType>
```

```
<participantType name="EchoForwarder">
  <portType name="Echoee" invdef="tns:Echoer"/>
  <portType name="Echoer" def="tns:Echoer"/>
</participantType>
```

The protocol specification from the *EchoServer2*'s perspective, could look something like (note that from *EchoServer2*'s perspective *EchoServer1* is it's client):

```
<protocolType name="Echo">
  <self type="tns:EchoServer"/>

  <participant name="EchoClient" type="tns:Client"/>

  <serviceInput operation="echo" portType="Echoer" participantType="Client"
          participantPortType="Echoee"/>
    <choice>
      <serviceOutput/>
      <serviceFault name="EchoException"/>
    </choice>
  </serviceInput>
</protocolType>
```

The protocol specification from the *EchoServer1*'s perspective, could look something like:

```
<protocolType name="Forward">
  <self type="tns:EchoForwarder"/>

  <participant name="Client" type="tns:EchoClient"/>

  <participant name="Echoer" type="tns:EchoServer"/>

  <serviceInput operation="echo" portType="Echoer" participant="Client"
          participantPortType="Echoee">
    <choice>
      <invokeOutput operation="echo" portType="Echoee" participant="Server"
              participantPortType="Echoer">
        <choice>
          <sequence>
            <invokeInput/>
            <serviceOutput/>
          </sequence>
          <sequence>
            <invokeFault name="EchoException"/>
            <serviceFault name="EchoException"/>
          </sequence>
        </choice>
      </invokeOutput>
      <serviceFault name="EchoException"/>
    <choice>
  </serviceInput>
</protocolType>
```

Or, if the behaviour of *EchoServer1* is not specified:

```
<serviceInput operation="echo" portType="Echoer" participant="Client"
        participantPortType="Echoee">
  <choice>
    <invokeOutput operation="echo" portType="Echoee" participant="Server"
            participantPortType="Echoer">
      <sequence>
        <choice>
          <invokeInput/>
          <invokeFault name="EchoException"/>
        </choice>
        <choice>
          <serviceFault name="EchoException"/>
          <serviceOutput/>
        </choice>
      </sequence>
    </invokeOutput>
    <serviceFault name="EchoException"/>
  </choice>
</serviceInput>
```

The protocol specification from the *Client*'s perspective, could look something like:

```
<protocolType name="Call">
  <self type="tns:EchoClient"/>

  <participant name="Echoer" type="tns:EchoServer"/>

  <invokeOutput operation="echo" portType="Echoee" participant="Server"
          participantPortType="Echoer">
    <choice>
      <invokeInput/>
      <invokeFault name="EchoException"/>
    </choice>
  </invokeOutput>
</protocolType>
```

Π-Calculus equivalent

(EchoServer2 Perspective)

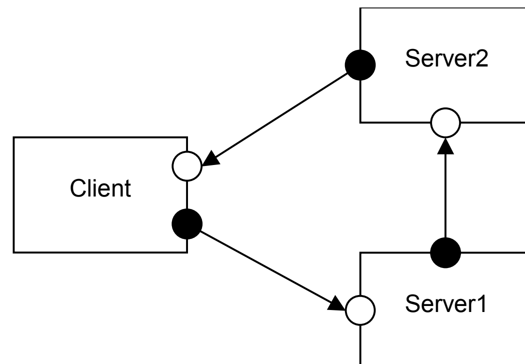
```
CE(EchoReq) . (CE<EchoRes> + CE<EchoException>)
```

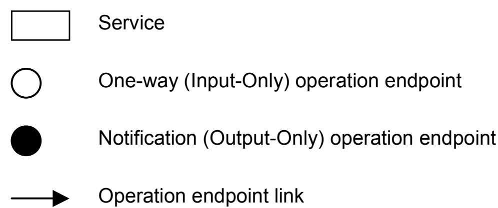
(EchoServer1 Perspective)

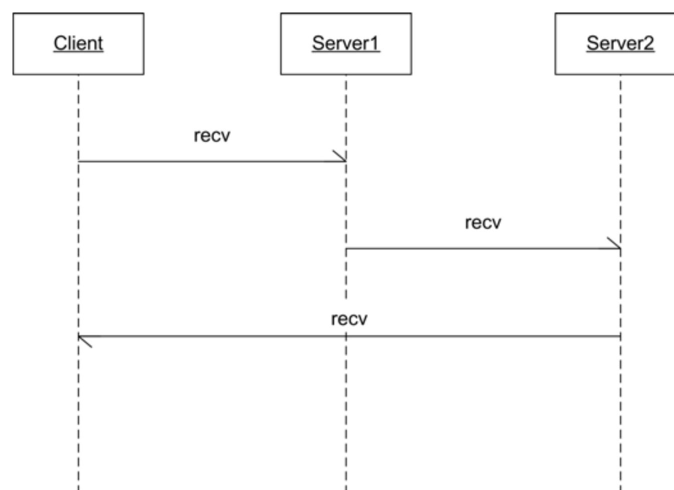
```
CE(EchoReq) . (E1E2<EchoReq> . (E2E1(EchoRes) . EC<EchoRes>) +
E2E1(EchoException) . EC<EchoException>)) + EC<EchoException>
```

5.7.3. Indirect Response Example

In this scenario, the *Client* sends an asynchronous message to *Server1*, which causes *Server1* to send a message to *Server2*, which in turn causes *Server2* to send a message to *Client*.







(a) Normal Termination

The port types involved in this scenario are:

```
<wsdl:portType name="ClientOpers">
  <wsdl:operation name="send">
    <wsdl:output message="tns:Message1"/>
  </wsdl:operation>
  <wsdl:operation name="recv">
    <wsdl:input message="tns:Message3"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="Server1Opers">
  <wsdl:operation name="recv">
    <wsdl:input message="tns:Message1"/>
  </wsdl:operation>
  <wsdl:operation name="send">
    <wsdl:output message="tns:Message2"/>
```

```
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="Server2Opers">
  <wsdl:operation name="recv">
    <wsdl:input message="tns:Message2"/>
  </wsdl:operation>
  <wsdl:operation name="send">
    <wsdl:output message="tns:Message3"/>
  </wsdl:operation>
</wsdl:portType>
```

This scenario contains three types of participants:

```
<participantType name="Client">
  <portType name="Opers" def="tns:ClientOpers"/>
</participantType>

<participantType name="Server1">
  <portType name="Opers" def="tns:Server1Opers"/>
</participantType>

<participantType name="Server2">
  <portType name="Opers" def="tns:Server2Opers"/>
</participantType>
```

The protocol specification from the *Client*'s perspective, could look something like:

```
<protocolType name="Request">
  <self type="tns:Client"/>

  <participant name="Server1" type="tns:Server1"/>

  <participant name="Server2" type="tns:Server2"/>

  <send operation="send" portType="Opers" participant="Server1"
      participantPortType="Opers"/>
  <receive operation="recv" portType="Opers" participant="Server2"
      participantPortType="Opers"/>
</protocolType>
```

The protocol specification from the *Server1*'s perspective, could look something like:

```
<protocolType name="Service">
  <self type="tns:Server1"/>

  <participant name="Client" type="tns:Client"/>

  <participant name="Server2" type="tns:Server2"/>

  <receive operation="recv" portType="Opers" participant="Client"
      participantPortType="Opers">
  <send operation="send" portType="Opers" participant="Server2"
      participantPortType="Opers">
</protocolType>
```

The protocol specification from the *Server2*'s perspective, could look something like:

```
<protocolType name="Response">
  <self type="tns:Server2"/>

  <participant name="Server1" type="tns:Server1"/>

  <participant name="Client" type="tns:Client"/>

  <receive operation="recv" portType="Opers" participant="Server1"
      participantPortType="Opers">
  <send operation="send" portType="Opers" participant="Client"
      participantPortType="Opers">
</protocolType>
```

Π-Calculus equivalent

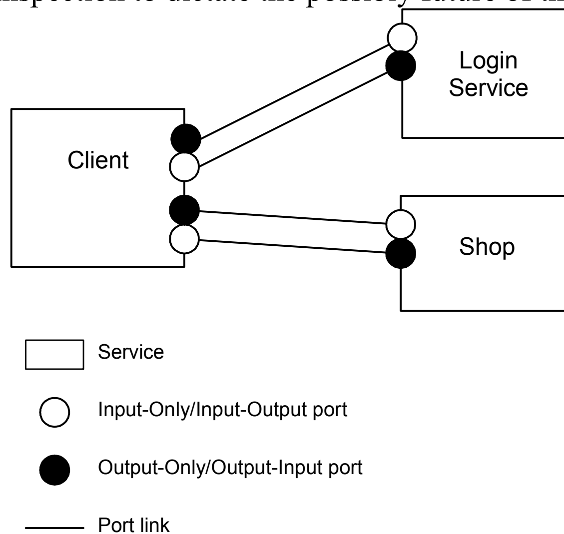
(Server1 Perspective)

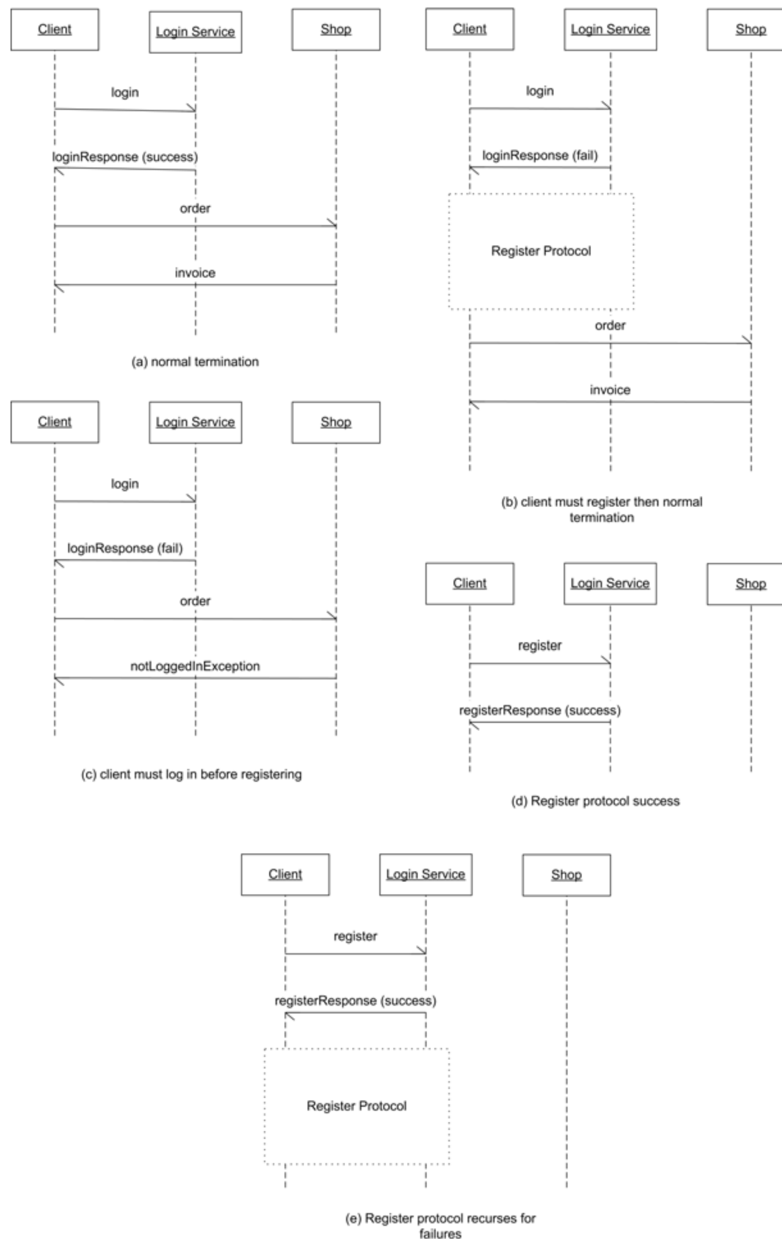
```
CS1(Message1) . S1S2<Message2>
```

(Server1 Perspective)
`S1S2(Message2) . S2C<Message3>`

5.7.4. Login and Shop Example

In this scenario, the *Client* send an asynchronous message to a login service in order to authenticate with a view to shopping. The *Login Service* will in response, send an asynchronous message to the client with the result of that login. This message can indicate either success or failure to login. If the message indicates failure, the client can send a message to the *register* operation of the *Login Service*. This login process can be repeated as many times as necessary to perform a successful registration.
Once the *Client* has authenticated it can send asynchronous messages to operations of the *Shop*, which in this simplified example do not fail. This example shows the need for message inspection to dictate the possibly future of the conversation.





The port types involved in this scenario are:

```
<wsdl:portType name="LoginServicePT">
  <wsdl:operation name="login" parameterOrder="text">
    <wsdl:input message="tns:LoginRequest"/>
    </wsdl:operation>
  <wsdl:operation name="register" parameterOrder="text">
    <wsdl:input message="tns:RegisterRequest"/>
    </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="ClientPT">
  <wsdl:operation name="loginResult" parameterOrder="text">
    <wsdl:input message="tns:LoginResponse"/>
    </wsdl:operation>
    <wsdl:operation name="registerResult" parameterOrder="text">
      <wsdl:input message="tns:registerResponse"/>
    </wsdl:operation>
  <wsdl:operation name="invoice" parameterOrder="text">
      <wsdl:input message="tns:invoice"/>
  </wsdl:operation>
    <wsdl:operation name="notLoggedIn" parameterOrder="text">
    <wsdl:input message="tns:notLoggedIn"/>
  </wsdl:operation>
</wsdl:portType>
```

```
<wsdl:portType name="ShopPT">
  <wsdl:operation name="order" parameterOrder="text">
    <wsdl:input message="tns:order"/>
  </wsdl:operation>
</wsdl:portType>
```

This scenario contains three types of participants:

```
<participantType name="Client">
  <portType name="Client" def="tns:ClientPT"/>
  <portType name="LoginService" invdef="LoginServiecPT"/>
  <portType name="Shop" invdef="ShoptPT"/>
</participantType>

<participantType name="LoginService">
  <portType name="LoginService" def="tns:LoginServicePT"/>
  <portType name="ClientLogin" invdef="ClientPT"/>
</participantType>

<participantType name="Shop">
  <portType name="Shop" def="tns:ShopPT"/>
  <portType name="ClientShop" invdef="ClientPT"/>
</participantType>
```

The protocol specification from the *LoginService's* perspective, could look something like:

```
<protocolType name="authenticate">
  <self type="tns:LoginService"/>
  <participant name="Client" type="tns:Client"/>
  <sequence>
    <receive operation="login" portType="LoginService" participant="Client"
participantPortType="ClientPT"/>
    <!-- either we send a loginResult and finish or we send a loginResult and then can
continue with register -->
      <choice>
      <send operation="loginResult" content="boolean(//success)" portType="ClientLogin"
participant="Client" participantPortType="ClientPT"/>
      <sequence>
        <send operation="loginResult" content="boolean(//failure)"
portType="ClientLogin" participant="Client" participantPortType="ClientPT"/>
        <protocol ref="register"/>
        </sequence>
    </choice>
  </sequence>
</protocolType>

<!-- Re-use of the protocol "register" -->

<protocolType name="register">
  <self type="tns:LoginService"/>
  <participant name="Client" type="tns:Client"/>
    <sequence>
      <receive operation="register" portType="LoginService" participant="Client"
participantPortType="ClientPT"/>
      <choice>
        <send operation="registerResult" content="boolean(//success)"
portType="ClientLogin" participant="Client" participantPortType="ClientPT"/>
      <!-- Recurse if the register operation fails -->
        <sequence>
        <send operation="registerResult" content="boolean(//failure)"
portType="ClientLogin" participant="Client" participantPortType="ClientPT"/>
        <protocol ref="register"/>
      </sequence>
       </choice>
   </sequence>
</protocolType>
```

The protocol specification from the *Shop's* perspective, could look something like:

```
<protocolType name="buy">
  <self type="Shop"/>
    <participant name="Client" type="tns:Client"/>
```

```
  <choice>
    <sequence>
      <receive operation="order" portType="Shop" participant="Client"
participantPortType="ClientPT"/>
      <send operation="invoice" portType="ClientShop" participant="Client"
participantPortType="ClientPT"/>
    </sequence>
    <sequence>
      <receive operation="order" portType="Shop" participant="Client"
participantPortType="ClientPT"/>
      <send operation="notLoggedIn" portType="ClientShop" participant="Client"
participantPortType="ClientPT"/>
    </sequence>
  </choice>
</protocolType>
```

The protocol specification from the *Client*'s perspective, could look something like:

```
<protocolType name="loginAndPurchase">
  <self type="Client"/>
  <participant name="LoginService" type="tns:LoginService"/>
  <participant name="Shop" type="tns:Shop"/>
  <sequence>
    <send operation="login" portType="LoginService" participant="LoginService"
participantPortType="LoginServicePT"/>
    <choice>
      <sequence>
        <receive operation="loginResult" content="boolean(//success)" portType="Client"
participant="LoginService" participantPortType="LoginServicePT"/>
        <send operation="order" portType="Shop" participant="Shop"
participantPortType="ShopPT"/>
        <receive operation="invoice" portType="Client" participant="Shop"
participantPortType="ShopPT"/>
      </sequence>
        <sequence>
          <receive operation="loginResult" content="boolean(//failure)"
portType="Client" participant="LoginService" participantPortType="LoginServicePT"/>
          <protocol ref="ClientRegister"/>
          <send operation="order" portType="Shop" participant="Shop"
participantPortType="ShopPT"/>
          <receive operation="invoice" portType="Client" participant="Shop"
participantPortType="ShopPT"/>
        </sequence>
      </choice>
    </sequence>
</protocolType>

<protocolType name="ClientRegister">
    <self type="tns:Client"/>
    <participant name="LoginService" type="tns:LoginService"/>
    <sequence>
      <send operation="register" portType="LoginService" participant="LoginService"
participantPortType="LoginServicePT"/>
      <choice>
        <receive operation="registerResult" content="boolean(//success)"
portType="Client" participant="LoginService" participantPortType="LoginServicePT"/>
        <!-- Recurse if the register operation fails -->
          <sequence>
        <receive operation="registerResult" content="boolean(//failure)"
portType="Client" participant="LoginService" participantPortType="LoginServicePT"/>
        <protocol ref="ClientRegister"/>
        </sequence>
    </choice>
    </sequence>
</protocolType>
```

Π-Calculus

(Shop Perspective)

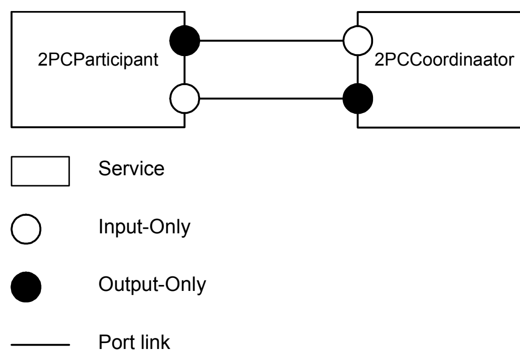
```
CS(order) . (SC<invoice> + CS<invoiceException>)
```

(LoginService Perspective)

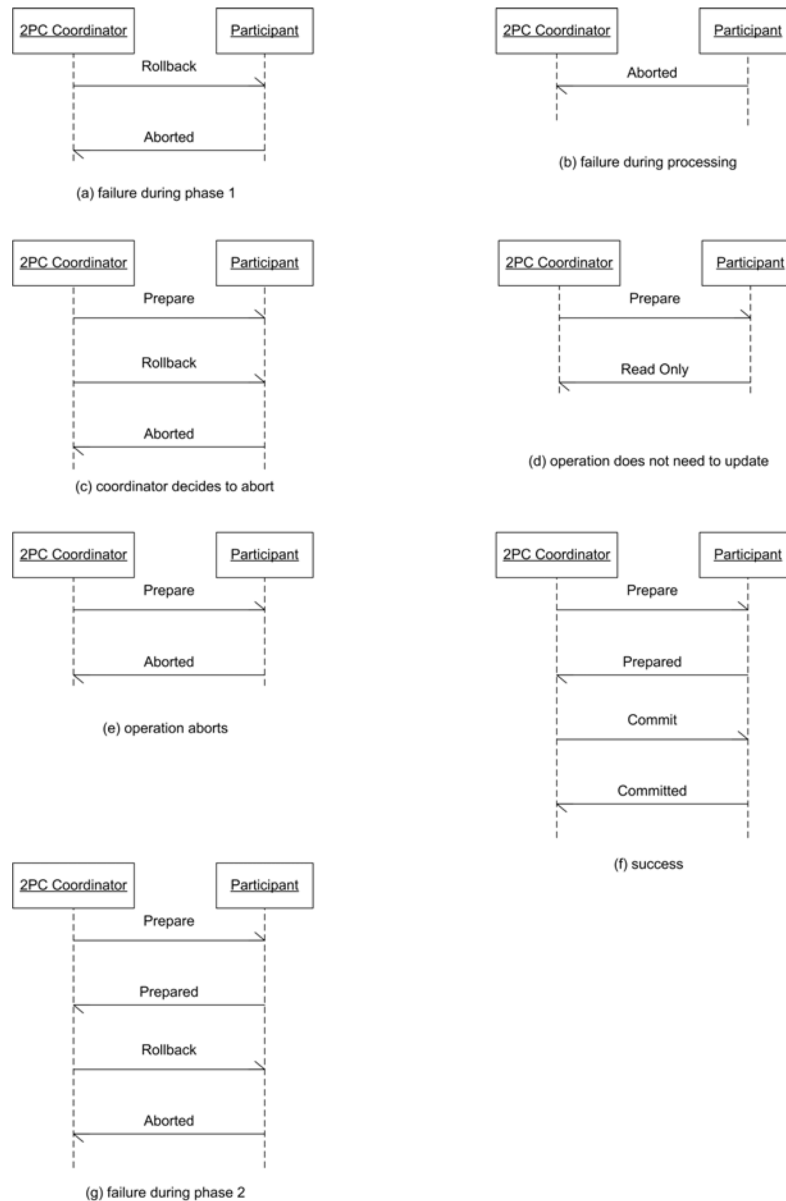
```
CLS(login) . LSC<loginResponse> . (0 + !(CLS(register) .
LSC(registerResponse)))
```

5.7.5. WS-Transaction Two Phase Commit

In this scenario, a 2PCommittable resource wishes to take part in a transaction according to the semantics specified in the WS-Transaction specification. The participant implements the 2PCParticipant interface and a 2PCCoordinator is used to control the transaction. Full details of the protocol are available at: http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/



UML Sequence diagrams for the possible conversations supported by the 2PC Coordinator are shown below:



There are two participants, the 2PCCoordinator and the 2PCParticipant

```
<participantType name="twoPCCoordinator">
  <portType name="twoPCCoordinator" def="tns:twoPCCoordinatorPT"/>
  <portType name="twoPCParticipant" invdef="tns:twoPCParticipantPT"/>
</participantType>

<participantType name="twoPCParticipant">
  <portType name="twoPCParticipant" def="tns:twoPCParticipantPT"/>
  <portType name="twoPCCoordinator" invdef="tns:twoPCCoordinatorPT"/>
</participantType>
```

The Conversation offered by the 2PCCoordinator could look like:

```
<protocolType name="authenticate">
  <self type="tns:twoPCCoordinator"/>
  <participant name="twoPCParticipant" type="tns:twoPCParticipant"/>
  <choice>
    <sequence>
      <send operation="rollback" portType="twoPCParticipant"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
      <receive operation="aborted" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
    </sequence>
    <receive operation="aborted" portType="twoPCCoordinator"
```

```
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
    <sequence>
      <send operation="prepare" portType="twoPCParticipant"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
      <choice>
        <sequence>
          <send operation="rollback" portType="twoPCParticipant"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
          <receive operation="aborted" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
        </sequence>
        <receive operation="read-only" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
        <receive operation="aborted" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
        <sequence>
          <receive operation="prepared" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
          <choice>
            <sequence>
              <send operation="commit" portType="twoPCParticipant"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
              <receive operation="committed" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
            </sequence>
            <sequence>
              <send operation="rollback" portType="twoPCParticipant"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
              <receive operation="aborted" portType="twoPCCoordinator"
participant="twoPCParticipant" participantPortType="twoPCParticipantPT"/>
            </sequence>
          </choice>
        </sequence>
      </choice>
    </sequence>
  </choice>
</protocolType>
```

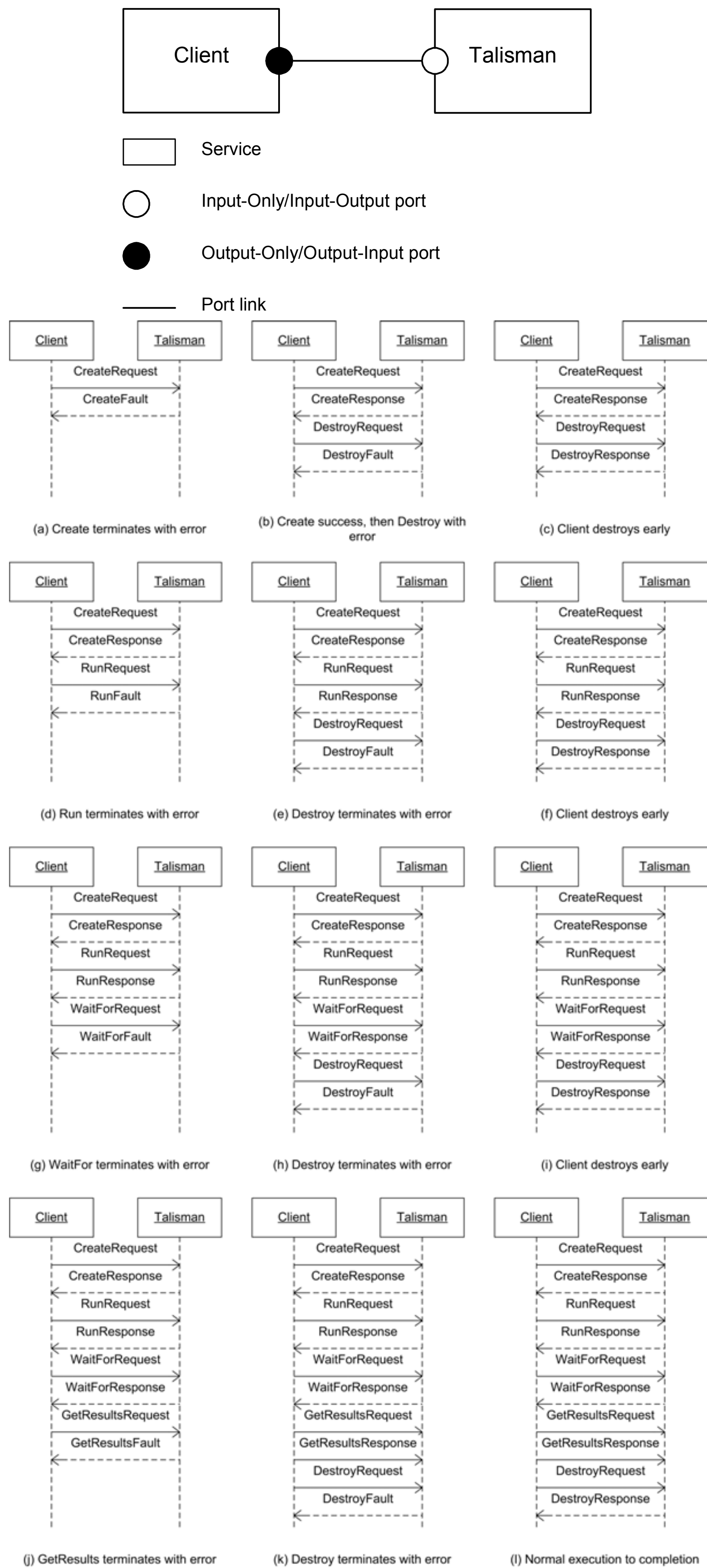
As this is just a two way conversation, the 2PCParticipant side would be an inverse of the 2PCCoordinator

(From 2PC Coordinator Perspective)

```
(<Rollback> . T . (Aborted)) +
(Aborted) +
(<Prepare> . (<Rollback> . T . (Aborted)) + (Read-only) + (Aborted) +
((Prepared) . T . ((<Commit> . (Committed)) + (<Rollback> .
(Aborted))))
```

5.7.6. Talisman Example

In this scenario, which is based on a well know Grid application, the Client may make invocations on the service in a specific order, as follows: First they must call ‘createJob’, if this is successful they may call ‘run’, following a successful call to ‘run’ they can call ‘waitForResults’, which blocks till results are available and finally they can call ‘getResults’. The ‘destroy’ operation may be called at any time after job creation, nothing may be called after ‘destroy’. If a fault occurs then they can not carry on with any operations with this job.



The port type involved in this scenario is:

```
<wsdl:portType name="Talisman">
  <wsdl:operation name="createJob" parameterOrder="in0">
    <wsdl:input message="intf:createJobRequest" name="createJobRequest"/>
    <wsdl:output message="intf:createJobResponse" name="createJobResponse"/>
    <wsdl:fault message="intf:SoaplabException" name="SoaplabException"/>
  </wsdl:operation>
  <wsdl:operation name="destroy" parameterOrder="in0">
    <wsdl:input message="intf:destroyRequest" name="destroyRequest"/>
    <wsdl:output message="intf:destroyResponse" name="destroyResponse"/>
    <wsdl:fault message="intf:SoaplabException" name="SoaplabException"/>
  </wsdl:operation>
  <wsdl:operation name="getResults" parameterOrder="in0">
    <wsdl:input message="intf:getResultsRequest" name="getResultsRequest"/>
    <wsdl:output message="intf:getResultsResponse" name="getResultsResponse"/>
    <wsdl:fault message="intf:SoaplabException" name="SoaplabException"/>
  </wsdl:operation>
  <wsdl:operation name="run" parameterOrder="in0">
    <wsdl:input message="intf:runRequest" name="runRequest"/>
    <wsdl:output message="intf:runResponse" name="runResponse"/>
    <wsdl:fault message="intf:SoaplabException" name="SoaplabException"/>
  </wsdl:operation>
  <wsdl:operation name="waitFor" parameterOrder="in0">
    <wsdl:input message="intf:waitForRequest" name="waitForRequest"/>
    <wsdl:output message="intf:waitForResponse" name="waitForResponse"/>
    <wsdl:fault message="intf:SoaplabException" name="SoaplabException"/>
  </wsdl:operation>
</wsdl:portType>
```

This scenario contains two types of participants:

```
<participantType name="Client">
  <portType name="User" invdef="tns:Talisman"/>
</participantType>

<participantType name="Server">
  <portType name="Provider" def="tns:Talisman"/>
</participantType>
```

The protocol specifications from the Server perspective could look something like:

```
<protocolType name="Destroy">
  <self type="tns:Server"/>
  <participant name="Client" type="tns:Client"/>
  <serviceInput operation="destroy" portType="Talisman" participant="Client"
participantPortType="User">
    <choice>
      <serviceOutput/>
      <serviceFault name="destroyFault"/>
    </choice>
  </serviceInput>
</protocolType>

<protocolType name="Seqret">
  <serviceInput operation="create" portType="Talisman" participant="Client"
participantPortType="User"/>
    <choice>
      <serviceFault name="createFault"/>
      <sequence>
        <serviceOutput/>
        <choice>
          <protocol ref="Destroy"/>
          <serviceInput operation="run" portType="Talisman" participant="Client"
participantPortType="User">
            <choice>
              <serviceFault name="runFault"/>
              <sequence>
                <serviceOutput/>
                <choice>
                  <protocol ref="Destroy"/>
                  <serviceInput operation="waitFor" portType="Talisman"
participant="Client" participantPortType="User"/>
                    <choice>
                      <serviceFault name="waitForFault"/>
```

```
                      <sequence>
                        <serviceOutput/>
                        <choice>
                          <protocol ref="Destroy"/>
                          <serviceInput operation="getResults" portType="Talisman"
participant="Client" participantPortType="User"/>
                            <choice>
                              <serviceFault name="getResultsFault"/>
                              <sequence>
                                <serviceOutput/>
                                <protocol ref="Destroy"/>
                              </sequence>
                            </choice>
                          </serviceInput>
                        </choice>
                      </sequence>
                    </choice>
                  </serviceInput>
                </choice>
              </sequence>
            </choice>
          </serviceInput>
        </choice>
      </sequence>
    </choice>
  </serviceInput>
</protocolType>
```

5.8. Schemas

XML Schema for WSDL extensibility elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://schemas.adapt.eu.org/service-protocol-defintion-
1.0/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://schemas.adapt.eu.org/service-protocol-defintion-1.0/"
elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xsd:element name="protocol" type="tns:ProtocolType"/>
   <xsd:complexType name="ProtocolType">
      <xsd:sequence>
         <xsd:element name="self" type="tns:SelfType"/>
         <xsd:element name="participant" type="tns:ParticipantType" minOccurs="0"
maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="type" type="xsd:QName"/>
   </xsd:complexType>
   <xsd:complexType name="SelfType">
      <xsd:sequence>
         <xsd:element name="portType" type="tns:PortTypeType" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="ParticipantType">
      <xsd:sequence>
         <xsd:element name="participant" type="tns:PortTypeType" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName"/>
   </xsd:complexType>
   <xsd:complexType name="PortTypeType">
      <xsd:attribute name="name" type="xsd:NCName"/>
      <xsd:attribute name="port" type="xsd:QName"/>
   </xsd:complexType>
</xsd:schema>
```

XML Schema for interaction constraints.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://schemas.adapt.org/interaction-constraints-
definition-1.0/" xmlns:tns="http://schemas.adapt.org/interaction-constraints-definition-
1.0/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
   <xsd:element name="interactionConstraints" type="tns:InteractionConstraintsType"/>
   <!-- Interaction constraints type -->
```

```
   <xsd:complexType name="InteractionConstraintsType">
      <xsd:sequence>
         <xsd:element name="participantType" type="tns:ParticipantTypeType"
minOccurs="0" maxOccurs="unbounded"/>
         <xsd:element name="protocolType" type="tns:ProtocolTypeType" minOccurs="0"
maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="optional"/>
   </xsd:complexType>
   <!-- Participant and protocol types -->
   <xsd:complexType name="ParticipantTypeType">
      <xsd:sequence>
         <xsd:element name="portType" type="tns:PortTypeType" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName" use="required"/>
   </xsd:complexType>
   <xsd:complexType name="PortTypeType">
      <xsd:attribute name="name" type="xsd:NCName" use="required"/>
      <xsd:attribute name="def" type="xsd:QName" use="optional"/>
      <xsd:attribute name="invdef" type="xsd:QName" use="optional"/>
   </xsd:complexType>
   <xsd:complexType name="ProtocolTypeType">
      <xsd:sequence>
         <xsd:element name="self" type="tns:SelfType"/>
         <xsd:element name="participant" type="tns:ParticipantType"
maxOccurs="unbounded"/>
         <xsd:group ref="tns:ActionType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:NCName" use="required"/>
   </xsd:complexType>
   <xsd:complexType name="SelfType">
      <xsd:attribute name="type" type="xsd:QName" use="required"/>
   </xsd:complexType>
   <xsd:complexType name="ParticipantType">
      <xsd:attribute name="name" type="xsd:NCName" use="required"/>
      <xsd:attribute name="type" type="xsd:QName" use="required"/>
      <xsd:attribute name="abstract" type="xsd:boolean" use="optional" default="false"/>
   </xsd:complexType>
   <xsd:complexType name="ProtocolRefType">
      <xsd:sequence>
         <xsd:element name="participant" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
               <xsd:attribute name="name" type="xsd:NCName"/>
               <xsd:attribute name="ref" type="xsd:QName"/>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="ref" type="xsd:QName" use="required"/>
   </xsd:complexType>
   <!-- Structuring action types -->
   <xsd:complexType name="SequenceType">
      <xsd:group ref="tns:ActionType" minOccurs="2" maxOccurs="unbounded"/>
   </xsd:complexType>
   <xsd:complexType name="ChoiceType">
      <xsd:group ref="tns:ActionType" minOccurs="2" maxOccurs="unbounded"/>
   </xsd:complexType>
   <xsd:complexType name="ParallelType">
      <xsd:group ref="tns:ActionType" minOccurs="2" maxOccurs="unbounded"/>
   </xsd:complexType>
   <xsd:complexType name="MultipleType">
      <xsd:group ref="tns:ActionType"/>
   </xsd:complexType>
   <xsd:simpleType name="NothingType">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value=""/>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- Asynchronous interaction types -->
   <xsd:complexType name="SendType">
      <xsd:attributeGroup ref="tns:InteractionAttributes"/>
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <xsd:complexType name="ReceiveType">
      <xsd:attributeGroup ref="tns:InteractionAttributes"/>
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <!-- Synchronous interaction types for the recipient side of the call -->
```

```
   <xsd:complexType name="InvokeOutputType">
      <xsd:group ref="tns:ActionType" maxOccurs="unbounded"/>
      <xsd:attributeGroup ref="tns:InteractionAttributes"/>
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <xsd:complexType name="InvokeInputType">
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <xsd:complexType name="InvokeFaultType">
      <xsd:attribute name="name" type="xsd:NCName" use="required"/>
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <!-- Synchronous interaction types for the initiating side of the call -->
   <xsd:complexType name="ServiceInputType">
      <xsd:group ref="tns:ActionType"/>
      <xsd:attributeGroup ref="tns:InteractionAttributes"/>
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <xsd:complexType name="ServiceOutputType">
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <xsd:complexType name="ServiceFaultType">
      <xsd:attribute name="name" type="xsd:NCName" use="required"/>
      <xsd:attributeGroup ref="tns:XpathAttributes"/>
   </xsd:complexType>
   <!-- Interaction attributes  -->
   <xsd:attributeGroup name="InteractionAttributes">
      <xsd:attribute name="operation" type="xsd:NCName" use="required"/>
      <xsd:attribute name="portType" type="xsd:NCName" use="required"/>
      <xsd:attribute name="participant" type="xsd:NCName" use="required"/>
      <xsd:attribute name="participantPortType" type="xsd:NCName" use="required"/>
      <xsd:attribute name="participantOperation" type="xsd:NCName" use="optional"/>
   </xsd:attributeGroup>
   <!-- Attributes for selecting content of messages -->
   <xsd:attributeGroup name="XpathAttributes">
      <xsd:attribute name="content" type="xsd:string" use="optional"/>
      <xsd:attribute name="participantBindingName" type="xsd:NCName" use="optional"/>
      <xsd:attribute name="participantBindingContent" type="xsd:string" use="optional"/>
   </xsd:attributeGroup>
   <!-- Action type -->
   <xsd:group name="ActionType">
      <xsd:choice>
         <xsd:element name="sequence" type="tns:SequenceType"/>
         <xsd:element name="choice" type="tns:ChoiceType"/>
         <xsd:element name="parallel" type="tns:ParallelType"/>
         <xsd:element name="multiple" type="tns:MultipleType"/>
         <xsd:element name="nothing" type="tns:NothingType"/>
         <xsd:element name="send" type="tns:SendType"/>
         <xsd:element name="receive" type="tns:ReceiveType"/>
         <xsd:element name="invokeOutput" type="tns:InvokeOutputType"/>
         <xsd:element name="invokeInput" type="tns:InvokeInputType"/>
         <xsd:element name="invokeFault" type="tns:InvokeFaultType"/>
         <xsd:element name="serviceInput" type="tns:ServiceInputType"/>
         <xsd:element name="serviceOutput" type="tns:ServiceOutputType"/>
         <xsd:element name="serviceFault" type="tns:ServiceFaultType"/>
         <xsd:element name="protocol" type="tns:ProtocolRefType"/>
      </xsd:choice>
   </xsd:group>
</xsd:schema>
```

References

[1] Web Services Description Language (WSDL) 1.1 (http://www.w3.org/TR/wsdl)

[2] R.Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, 1999

[3] xCBL Order Management Use Case (http://www.xcbl.org)

[4] XML Path Language (XPATH) (http://www.w3.org/TR/xpath)

[5] Web Service Transactions (WS-T)
(http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/)

[6] Web Service Coordination (WS-C)
(http://www-106.ibm.com/developerworks/library/ws-coor/)

[7] Business Process Execution Language for Web Service (BPEL4WS)
(http://www.ibm.com/developerworks/library/ws-bpel/)

[8] Web Service Choreography Interface (WSCI) (http://www.w3.org/TR/wsci/)

[9] Web Service Conversation Language (WSCL) 1.0 (http://www.w3.org/TR/wscl10/)

[10] Web Services Choreography Working Group, web site
(http://www.w3.org/2002/ws/chor/)

[11] W3C XML Schema Part 1: Structures (http://www.w3.org/TR/xmlschema-1/)

[12] W3C XML Schema Part 2: Data-types (http://www.w3.org/TR/xmlschema-2/)

[13] ANSI X3.135-1992, American National Standard for Information
Systems – Database Language – SQL, Nov 1992.

[14] A. ShaikAli, O. F. Rana, R. Al-Al, D. W. Walker. UDDIe: An Extended Registry for Web Services.

[15] OASIS. UDDI Specification. Version 3.
http://www.uddi.org/specification.html

[16] OASIS. Using WSDL in a UDDI Registry. Version 1.08. Nov. 2002.

[17] OASIS. Business Transaction Protocol (BTP).

[18] BEA, IBM, Microsoft. Web Service Transaction (WS-Transaction), Aug. 2002.

[19] Arjuna, Oracle, Sun. Web Service Coordination Activity Framework (WS-CAF), Aug. 2003.

[20] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3):463-492, July 1990.

[21] Farrell and Kreger. Web services management approaches, IBM Systems Journal vol 41, n. 2, 2002.

[22] Oracle. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7, July 1995.

[23] OMG. CORBA Object Transaction Service (OTS).

[24] Alejandro P. Buchmann, M. Tamer Özsu, Dimitrios Georgakopoulos, Frank Manola: A Transaction Model for Active Distributed Object Systems. Database Transaction Models for Advanced Applications 1992, pp. 123-158.

[25] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, Abraham Silberschatz: A Transaction Model for Multidatabase Systems. ICDCS 1992, pp. 56-63.

[26] Aidong Zhang, Marian H. Nodine, Bharat K. Bhargava, Omran A. Bukhres: Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. SIGMOD Conference 1994, pp. 67-78.

[27] Victor, B and Moller, F. The Mobility Workbench: A tool for the Π-Calculus. http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-285/

[28] Deliverable 7: Composition Language. Adapt Project Deliverable, September 2003