ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Composition Language

Deliverable Identifier: D7
Delivery Date: 19th September 2003
Classification: Public Circulation
Authors: Simon Woodman, Santosh Shrivastava, Stuart Wheater, Doug Palmer
Document version: 1.0 17th September 2003

Contract Start Date: 1st 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. Language Overview 3
3. Language Reference 5
4. Formal Mapping to π-Calculus 13
5. Examples 14
6. Schema 18
7. References 20

Appendix A: A system for Distributed Enactment of Composite Web Services 22

1 Introduction

A composition language can be used to specify the structure and properties of a process. The composition language to be described here has been specifically designed to express process composition and inter-task dependencies of fault-tolerant distributed applications whose executions could span several autonomous organizations and arbitrarily large durations. "*Tasks*" are application specific units of computation and equate to the invocation of a web service. The work is motivated by the observation that an increasingly large number of distributed applications are constructed by composing them out of existing applications, which are executed in a heterogeneous environment. The resulting applications can be very complex in structure, containing many notification and dataflow dependencies between their constituent applications. Furthermore, the execution of such an application may take a long time to complete (days, weeks or even months), and may contain long periods of inactivity, often due to the constituent applications requiring user interventions. In a distributed environment, it is desirable that long running applications have support for fault-tolerance and dynamic reconfiguration: machines may fail, services may be moved or withdrawn and application requirements may change. In such an environment it is essential that the structure of applications can be modified dynamically (during execution) to reflect these changes.

The execution of an application is modelled as the execution of a collection of interdependent *tasks* (*activities*). A task represents a unit of work to be done in the form of an invocation of a web service (e.g., a request to book a hotel room). A "*process*" is termed the composition of one or more tasks and other processes. Each process is regarded as a single logical unit. Fig. 1 depicts the inter-task dependencies of four tasks (t_1, ..., t_4); t_2 and t_3 start once t_1 finishes and t_4 starts after both t_2 and t_3 have finished. A dependency could be just a *notification* dependency (shown by a dotted arc, indicating that t_2 can start only after t_1 has terminated) or a *dataflow* dependency (shown by a solid arc, indicating that, say t_3 needs input data from t_1). There follows a discussion of the four requirements of the model: fault-tolerance, dynamic reconfiguration, modularity and interoperability.

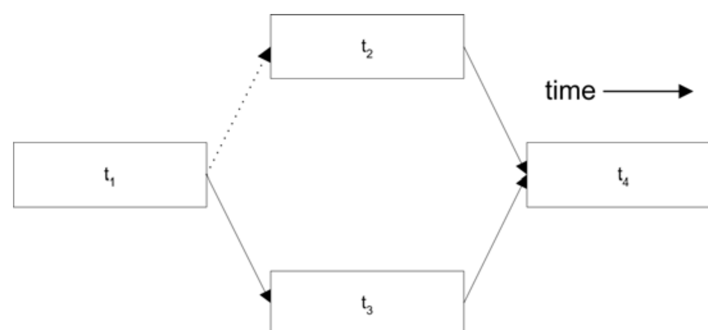


Figure 1, Inter task dependencies.

Fault-tolerance: The composition language offers application level fault tolerance in the form of alternate dependencies. Processes can be structured such that a task's input can come from multiple sources. This gives a degree of fault tolerance through redundancy. In our model compensatory tasks can be used to recover from situations where a task terminates by producing a fault message. To ensure processing continues despite processor failures our model allows the fragmenting of a process and its coordination

from a number of different hosts. A finite number of network related failures can be tolerated by retransmitting data between these hosts.

Dynamic Reconfiguration: A long running application is likely, at some point during its execution, to encounter changes in the environment within which it is executing. As stated earlier, these environmental changes could include machine and network related failures, services being moved or withdrawn, or even the application's functional requirements being changed. Mechanisms that will allow applications to change their internal structures to ensure forward progress are therefore required. It should be therefore possible to change the structure of a running application by adding/deleting tasks, notifications and dependencies.

Modularity: A process definition should enforce locality of modifications; only the parts of the script directly affected by a change should need changing. For example, adding an additional input dependency to a task should only affect the script for that task. Further, the composition language should provide flexible means for specifying the composition of a task in terms of other primitive tasks.

Interoperability: It should be possible to compose an application out of component applications in a uniform manner, irrespective of the programming languages in which the component applications have been written and the operating systems of the host platforms. Web services aid interoperability due to the standardised protocols used to communicate with them. SOAP [1] is used as a protocol to access web services, whose interfaces are described using WSDL [2] and can be located in a UDDI repository [3]. As long as the web services used conform to the specifications, the implementation of these services is irrelevant.

2 Language Overview

There are several industry led efforts aimed at specifying composition languages for web services, however all of these language taken a centralised view of composition and subsequent execution. For example the use of shared variables makes it very difficult to coordinate the execution in a distributed manner. Also, many of these languages specify complicated control flow mechanisms, making it difficult to analyse such compositions. This composition language has been developed with both of these drawbacks in mind, it contains elements to allow distributed composition and the simple data flow sequencing model has been mapped to π-calculus to allow analysis of compositions.

The composition language has been designed to allow the specification of the structure of applications at a level of abstraction which allows the composite service designer to concentrate on ensuring the correct functional behaviour of the workflow application, even in the presence of failures. Fault tolerance requirements of applications have been split into the requirements at the application level itself and at the system level (execution environment). The composition language provides notations and structures for meeting modularity and application level fault-tolerance requirements, whereas the execution environment is responsible for meeting system level fault tolerance. Meeting interoperability and dynamic reconfiguration requirements are also the responsibility of the execution environment. There follows a description of main features of the composition language and the execution environment and then a comprehensive discussion of the language constructs and their use.

There is both a graphical as well as a textual representation of the composition. A graphical representation of a task is given in fig. 2. It depicts a task (called task) that has one input set (I_1) with two data parts (i_1 and i_2). These correspond to the messages and parts defined in the WSDL document describing the service. The input sets must have all of its input parts available (input dependencies satisfied) before the task can start. A task terminates in one of the named output states (called *outcomes*). One of these outcomes is considered a *normal* outcome and all others are considered fault outcomes. In figure 2, O_1 represents an output message and F_1 represents a fault message. These terms are analogous to the web service returning an output or fault message as described in WSDL. Each outcome of a task has a distinct set of parts, which can be used as input objects by subsequent tasks or output objects by composing tasks. The output message in figure 2 has two named parts: o_1 and o_2 with respectively.

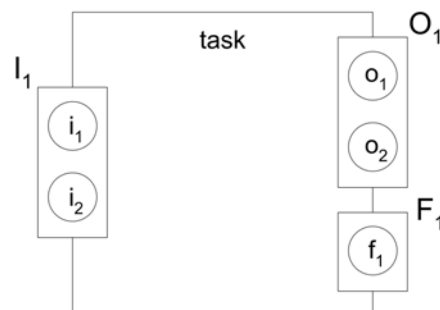


Figure 2, a task.

Process Definition: The language allows the definition of processes in two ways:

- In-line – this refers to the case when an anonymous process is defined within another process for the purposes of modularity. It is not possible to instantiate this process without the parent process being instantiated and it cannot be referenced from other processes.
- Out-of-line – out-of-line definition allows a designer to re-use process definitions which have been defined in a separate document. These processes can be referred to from within another process or instantiated on their own. This structure aids code re-use.

Instantiation Time: There are also two times at which the designer of a process definition can choose to instantiate the tasks and processes within it:

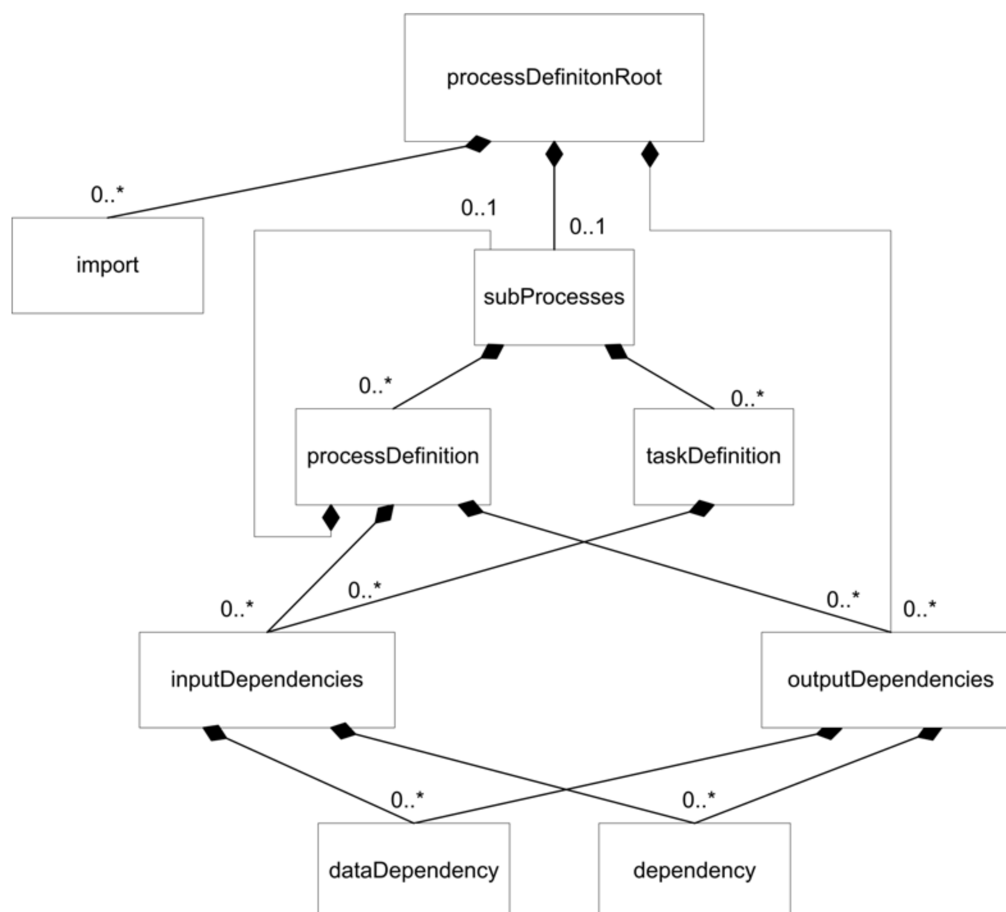
- Early – when early (traditional) instantiation is used, all of the tasks, processes and the sub processes constituent tasks etc. are loaded into the execution environment and initialised when the parent process is instantiated. This leads to a more static system which is easier to reason about but more difficult to modify.
- Late – late instantiation results in the tasks and sub processes of a process not being instantiated until they are able to run, i.e. when all of their input dependencies are satisfied. Late instantiation implies that only those parts of large process definitions which are needed will be instantiated, which allows more efficient use of resources. It also allows the designer to perform recursion. This happens when a late instantiated process refers to itself; when tasks within the process terminate with a particular outcome it will instantiate itself again and recurse. This gives a great deal of flexibility to the process designer.

Inter-task Dependencies: As described above, the language is structured in terms of tasks, i.e. an invocation of a web service and processes which are compositions of tasks and other processes. The control structure of the processes is described in terms of dependencies between those tasks and processes. Dependencies can control when a task is executed, by using input dependencies and when a process completes, by using output dependencies. Input dependencies describe when a task can start execution, either in terms of what other tasks have started/completed, or in terms of where the input data needs to come from in order for the task to start. Output dependencies describe how the output of a process is "built" from the output of the constituent tasks.

Execution Environment: The execution environment provides facilities to enable sets of inter-related tasks and processes forming an application to be executed and supervised in a dependable manner. The execution environment is described further in a paper shown in Appendix A.

3 Language Reference

This section is intended to provide a comprehensive reference to the composition language, beginning by showing the structure of the language with a UML Class diagram and proceeding to a discussion of the individual language constructs.



3.1 ImportType

The `importType` allows the user to import a schema and namespace into the process definitions document. When another processDefinition is imported, the user can refer to elements defined in that processDefinition using the namespace defined.

```
<xsd:complexType name="ImportType">
   <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
   <xsd:attribute name="location" type="xsd:anyURI" use="required"/>
</xsd:complexType>
```

The `ImportType` has no sub elements.

The `ImportType` has the following attributes:

- `namespace` - the namespace to refer to the processDefinition by.
- `location` - the location of the processDefinition to import.

Usage would be:

```
<import name="myImport" location="http://mydomain/myImportDoc"/>
```

3.2 ProcessDefinitionRootType

The `ProcessDefinitionRootType` is the type at the root of the process definition. The designer of the service must specify the composition in terms of the subprocesses which compose the process and how the output of the composition is formed from the subprocesses.

```
<xsd:complexType name="ProcessDefinitionRootType">
  <xsd:sequence>
    <xsd:element name="import" type="tns:ImportType" minOccurs="0"
maxOccurs="unbounded"/>
    <xsd:element name="subProcesses" type="tns:SubProcessesType" minOccurs="0"/>
    <xsd:element name="outputDependencies" type="tns:DependencyListType" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="tns:ActionAttributes"/>
  <xsd:attribute name="targetNamespace" type="xsd:anyURI"/>
</xsd:complexType>
```

The `ProcessDefinitionRootType` has the following sub elements:

- `import` – the processDefinition to import as well as the associated namespace. Described fully in Section 3.1.
- `subProcesses` – a list of the sub processes (`tasks` and `processes` which the service is composed of.
- `outputDependencies` – a list of dependencies which will build the output message of the composed service. This is described further in Sections 3.6 – 3.8

The `ProcessDefinitionRootType` has the following attributes:

- `targetNamespace` – the targetNamespace of the process definition
- `ActionAttributes` – Described fully in Section 3.9, these attributes describe properties of the service such as the portType and operation.

Usage would be:

```
<processDefinitionRoot>
  <import … />
  <subProcesses>
```

```
    …
  </subProcesses>
  <outputDependencies>
    …
  </outputDependencies>
</processDefinitionRoot>
```

3.3 ProcessDefinitionType

The `ProcessDefinitionType` is used to define processes which are themselves composite. A process can be composed of other `processes` and `tasks` which are the lowest level and equate to an invocation of a web service. A `process` is composed of the `inputDependencies` which must be satisfied for it to be executed; the `subprocesses` which compose it; the `outputDependencies` which construct the output of the composition. It is possible to define a process both *in-line* and *out-of-line*. If in-line definition is used, the `subProcesses` element contains the definition of the composition. An alternative to defining anonymous processes is to reference one which has been defined out-of-line in another document. To achieve this the definition attribute should be used and no subprocesses element should be used. The definition attribute should be a qualified reference to a process defined as a top level element in another document.

A process can use either early or late instantiation indicated by an attribute defined in the `ActionAttributes` group. This gives the possibility of either instantiating the process when the parent process is instantiated, or when the process's input dependencies are satisfied. The latter case is discussed further in [4] where they are referred to as *Genesis Tasks*. Late instantiation of a process allows recursion in a process definition as a process can instantiate another instance of itself if it completes in a particular state. A process should not refer to itself in a recursive manner if `lateInstantiation` is set to false. This is considered an error in the process definition and will be detected at validation time. Another benefit of late instantiation is that it allows only those parts of a (potentially large) composition which are needed to be loaded into memory. This allows resource usage on the server to be minimised.

```
<xsd:complexType name="ProcessDefinitionType">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="inputDependencies" type="tns:DependencyListType"
        minOccurs="0"/>
      <xsd:element name="subProcesses" type="tns:SubProcessesType" minOccurs="0"/>
      <xsd:element name="outputDependencies" type="tns:DependencyListType"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:choice>
  <xsd:attributeGroup ref="tns:ActionAttributes"/>
  <xsd:attribute name="definition" type="xsd:QName"/>
</xsd:complexType>
```

The `ProcessDefinitionType` has the following sub elements:

- `inputDependencies` – a list of the `dependencies` and `dataDependencies` which must be satisfied for the process to be executed
- `subProcesses` – a list of the sub processes (tasks and processes) which the service is composed of.

- `outputDependencies` – a list of `dependencies` which will build the output message of the composed service. This is described further in Sections 3.6 – 3.8

The `ProcessDefinitionType` has the following attributes:
- `ActionAttributes` – Described fully in Section 3.9, these attributes describe properties of the service such as the `portType` and `operation`.
- `definition` – if subProcesses are not specified, this attribute must be present and must reference a `processDefinitionRootType` defined in another document. The other document should be imported using the `import` element and have the correct namespace defined.

Usage would be:

```
<processDefinition lateInstantiation=”false” logicalHost=”a”>
  <inputDependencies>
    …
  </inputDependencies>
  <subProcesses>
    …
  </subProcesses>
  <outputDependencies>
    …
  </outputDependencies>
</processDefinition>
```

Or alternatively:

```
<processDefinition definition="ns2:process2"/>
```

3.4 TaskDefinitionType

The `TaskDefinitionType` is used to define an action which invokes a actual web service as part of the composition. It is the lowest level action element within a composition and can be considered as a *black box*. The task must specify under what situations it can be executed using the `inputDependencies` element and specify how to invoke the service using the `port` attribute.

```
<xsd:complexType name="TaskDefinitionType">
  <xsd:choice>
    <xsd:element name="inputDependencies" type="tns:DependencyListType"
      minOccurs="0"/>
  </xsd:choice>
  <xsd:attributeGroup ref="tns:ActionAttributes"/>
  <xsd:attribute name="port" type="xsd:QName"/>
</xsd:complexType>
```

The `TaskDefinitionType` has the following sub elements:
- `inputDependencies` – a list of the `dependencies` and `dataDependencies` which must be satisfied for the task to be executed

The `TaskDefinitionType` has the following attributes:
- `ActionAttributes` – Described fully in Section 3.9, these attributes describe properties of the service such as the `portType` and `operation`.
- `port` – the qualified name of the port defining the service in the associated WSDL document.

Usage would be:

```
<taskDefinition name="myTask" portType="myTaskPT"
  operation="myOperation" logicalHost="ns2:server2">
  <inputDependencies>
    …
  </inputDependencies>
</taskDefinition>
```

3.5 SubProcessesType

The `SubProcessesType` is used within a `processDefinition` to list the constituent tasks and processes which it is composed of. As described in Section 3.3, if the `subProcesses` are not defined within the `processDefinition`, the definition of the process should be referenced using the definition attribute and defined in a separate document.

```
<xsd:complexType name="SubProcessesType">
  <xsd:sequence>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="taskDefinition" type="tns:TaskDefinitionType"/>
      <xsd:element name="processDefinition" type="tns:ProcessDefinitionType"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

The `SubProcessesType` has the following sub elements:

- An unbounded set of `task` and `process` definitions which specify the composition of the process.

The `SubProcessesType` has no attributes.

Usage would be:

```
<subProcesses>
  <taskDefinition … />
  …
  <processDefinition …>
    … <!—An inline process definition -->
  </processDefinition>
</subProcesses>
```

3.6 DependencyType

The `DependencyType` type is used to describe the situation where one `task` or `process` is dependant on another one. That is, a task or process cannot begin executing until another task is in a particular state. For instance, a task has terminated by producing a fault message, or a process has begun execution because its input dependencies are fulfilled.

The *source* and *sink* of the dependency must be specified using the `DependencyAttributes` described in Section 3.10. These allow a task to specify precisely the source of the `dependency` including particular outcomes. It is possible to attach a priority to the `dependency` which is used to determine what should happen when alternate dependencies become available simultaneously.

```
<xsd:complexType name="DependencyType">
  <xsd:attributeGroup ref="tns:DependencyAttributes"/>
</xsd:complexType>
```

The `DependencyType` has no sub elements.

The `DependencyType` has the following attributes:

- `DependencyAttribute` group – this group is explained fully in Section 3.10

Usage would be:

```
<dependency source="ATask" sourceMessageType="output" priority="10"/>
```

3.7 DataDependencyType

The `DataDependencyType` is used to model the situation when a *down-stream* task or process requires data from and *up-stream* task or process in order to start or complete execution. Normally, down-stream tasks require data which is the output of up-stream tasks to start execution. However, processes must also use `DataDependencies` to generate their output from the output of their constituent `tasks` and `processes`.

A `DataDependency` can reference either a `wsdl:part` of a message, or the entire message. The latter case is the default if the `partName` attributes are not specified.

A `DataDependency` implies that there is also a `Dependency` present as the data cannot be available until the up-stream task is in the required state.

```
<xsd:complexType name="DataDependencyType">
  <xsd:attributeGroup ref="tns:DependencyAttributes"/>
  <xsd:attribute name="sourcePartName" type="xsd:string" use="optional"/>
  <xsd:attribute name="sinkPartName" type="xsd:string" use="optional"/>
</xsd:complexType>
```

The `DataDependencyType` has no sub elements.

The `DataDependencyType` has the following attributes:

- `DependencyAttribute` group – this group is explained fully in Section 3.10
- `sourcePartName` – the `wsdl:part` of the `message` which is the source of the data
- `sinkPartName` – the `wsdl:part` of the `message` which is the sink of the data

Usage would be:

```
<dataDependency source="ATask" sourceMessageType="output" sinkMessageType="input"/>
```

Or alternatively:

```
<dataDependency source="ATask" sourceMessageType="output" sourcePartName="partA"
sinkMessageType="input" sinkPartName="partB"/>
```

3.8 DependencyListType

The DependencyListType is a utility type which allows both input and output dependencies to be specified using the same type. It contains a list of Dependency and DataDependency elements.

```
<xsd:complexType name="DependencyListType">
  <xsd:sequence>
    <xsd:element name="dependency" type="tns:DependencyType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="dataDependency" type="tns:DataDependencyType" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

The DependencyListType has the following sub elements:

- a sequence of DependencyType and DataDependencyType elements to define the execution order restrictions.

The DependencyListType has no attributes.

3.9 ActionAttributes

The ActionAttributes are used to define the action which a task or process refers to. Each action can be considered to be a single *unit of work* and must be uniquely identified within the processDefinition, i.e. the document. The ActionAttributes have subtly different semantics depending on where they are used:

- A Task uses the portType and operation attributes to specify which abstract operation must be invoked from the appropriate service (as defined by the WSDL document with the same namespace).
- A top level ProcessDefinitionRoot uses the portType and operation to specify which operation has been invoked on the service to cause the execution of this composition.
- An in-line processDefinition should not specify either the portType or operation attribute as the process does not necessarily refer to an operation exposed by a web service.

The lateInstantiation attribute specifies whether the task or process should be instantiated at the same time as the parent process or not. If the task or process is not instantiated at the same time as the process it is instantiated when the input dependencies are satisfied. This allows the composition to include conditional looping and only those parts of a (potentially large) process which are needed must be loaded into memory.

The logicalHost attribute is used to define which host is responsible for coordinating the execution of the task or process if the composition is fragmented over multiple hosts. This is a deployment attribute and the abstract hostname is bound to a concrete host using the deployment descriptor. A full discussion of the deployment time options is beyond the scope of this document.

```
<xsd:attributeGroup name="ActionAttributes">
  <xsd:attribute name="name" type="xsd:QName" use="required"/>
  <xsd:attribute name="portType" type="xsd:QName" use="optional"/>
  <xsd:attribute name="operation" type="xsd:NCName" use="optional"/>
  <xsd:attribute name="lateInstantiation" type="xsd:boolean" use="optional"
    default="false"/>
```

```
  <xsd:attribute name="logicalHost" type="xsd:QName" use="required"/>
</xsd:attributeGroup>
```

3.10 DependencyAttributes

The `DependencyAttribute` group is used to uniquely identify the source and sink of the inter-task dependencies. As the dependencies are child elements of the down-stream task or process, it is only necessary to define the up-stream process or task directly. However, both the source and sink message types and names must be specified if there is the possibility of ambiguity. For instance, as a web service can only have one input message, if the `messageType` is input the `messageName` need not be specified. In fact, the `messageName` need only be used to differentiate between different types of `faults` which a service can terminate with. The designer of the composition is required to use enough of the attributes to uniquely identify the source and sink of the dependency.

The `priority` of a dependency determines what will happen if alternate sources of a dependency become available simultaneously. The value of the priority attribute must be a positive integer. The dependency which has the highest value will be selected. The behavour when two dependencies have the same priority value is undefined. Priority is only an issue for `dataDependencies`, where the highest available priority `dataDependency` will be used at the time when the input/output/fault message is completely available.

```
<xsd:attributeGroup name="DependencyAttributes">
  <xsd:attribute name="source" type="xsd:QName" use="required"/>
  <xsd:attribute name="sourceMessageType" type="tns:MessageTypeType" use="required"/>
  <xsd:attribute name="sourceMessageName" type="xsd:string" use="optional"/>
  <xsd:attribute name="sinkMessageType" type="tns:MessageTypeType" use="optional"/>
  <xsd:attribute name="sinkMessageName" type="xsd:string" use="optional"/>
  <xsd:attribute name="priority" type="xsd:integer" use="optional" default="0"/>
</xsd:attributeGroup>
```

3.11 MessageTypeType

The `MessageTypeType` defines an enumeration of the messageTypes allowed in WSDL. This results in all messageTypes for the dependencies being forced to be either

- `input`
- `output`
- `fault`

```
<xsd:simpleType name="MessageTypeType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="input"/>
    <xsd:enumeration value="output"/>
    <xsd:enumeration value="fault"/>
  </xsd:restriction>
</xsd:simpleType>
```

4 Formal mapping to pi-calculus

To ensure that the composition language is both complete and precise it has been mapped onto π-calculus. π-calculus is a formal notation for modelling and analysing

properties of mobile communicating systems. A communicating system in π-calculus is a collection of processes exchanging messages via named channels. In [5] Milner proves that communicating systems can be modelled 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.

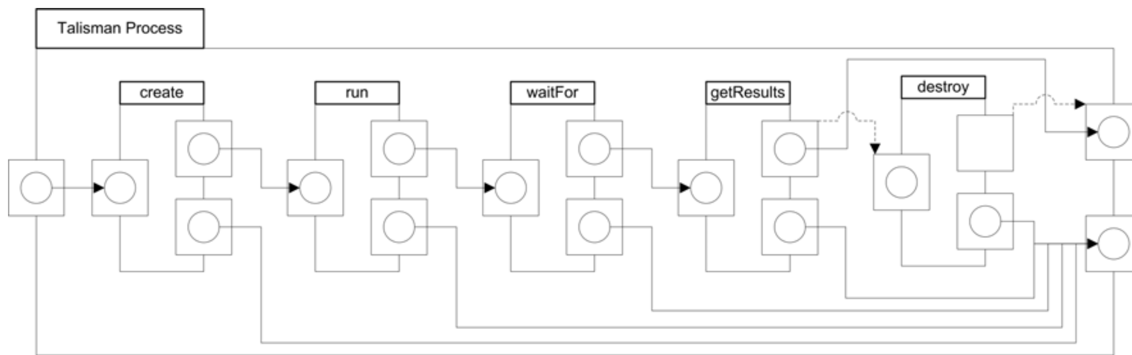
The first four primitives deal with the composition of processes, the final two primitives deal with communication between processes.

As well as proving that the composition language has a good semantic base, a π-calculus mapping allows formal reasoning about a composition. π-calculus gives elegant mechanisms for dealing with mobility of processes, this should help when dealing with changes to compositions. The equivalence relationships in π-calculus allow verification that two π-calculus expressions are equal. This serves a dual purpose; firstly it allows checking to ensure that a given composition respects a set of π-calculus based sequencing constraints, such as those as described in [6], secondly it allows the verification as to whether a set of sub-compositions still represents an original composition. This is useful when a composition is distributed across a number of controlling nodes.

In mapping the composition language to π-calculus a web service is regarded as a process to be executed. The input message used to invoke the service is regarded as the only input to the process. The output and fault messages generated by a service are regarded as the output from the process. Hence a web service 'A' with an input message 'ai', an output message 'ao' and a fault message 'af', could be modelled with the π-calculus as x(ai).A.(y<ao> + z<af>), where x,y,z are the channels over which the messages are sent. As the composition language does not talk in terms of individual web services and their associated input and output messages the π-calculus mapping is slightly more complicated. The web service is identified by attributes of the <TaskDefinition> tag, the port, portType and operation attributes combine to identify the associated service. The input message is derived from either all or part of other messages in the composition; the sourceMessageType, sourceMessageName, sourcePartName attributes of dependencies are used to identify the data of interest. Similarly, a dependency on an output or fault message generated by the execution of the web service might only need part of the message. Attributes of the dependencies also identify the links or channels between processes; the source attribute identifies the task at the sending end of a channel and the recipient end of the channel is the task that defines the dependency.

5 Examples

5.1 Talisman Application



The talisman example shown in the diagram above is a simple process consisting of five tasks executed in the following order – create, run, waitFor, getResults, destroy. The initial input data is passed to the create method, if create is successful then its output is passed to run as input. Then run is executed, if successful its results are passed as input to waitFor. Next getResults is executed; if successful its results are passed as output of the whole process. Finally the destroy operation is invoked, if successful then the process is allowed to terminate successfully. If any of the operations return a fault then the whole process terminates with a fault.

The WSDL for the services is shown below:

```
<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>
```

The process definition for this process is shown below:

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

<processDefinition name="talismanProcess" portType="TalismanProcessPT"
  operation="talismanProcess" xmlns="http://schemas.adapt.org/process-definition-2.1/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <subProcesses>
    <taskDefinition name="createTask" portType="Talisman" operation="create">
```

```
      <inputDependencies>
        <dataDependency source="talismanProcess" sourceMessageType="input"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="runTask" portType="Talisman" operation="run">
      <inputDependencies>
        <dataDependency source="createTask" sourceMessageType="output"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="waitForTask" portType="Talisman" operation="waitFor">
      <inputDependencies>
        <dataDependency source="runTask" sourceMessageType="output"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="getResultsTask" portType="Talisman" operation="getResults">
      <inputDependencies>
        <dataDependency source="waitForTask" sourceMessageType="output"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="destroyTask" portType="Talisman" operation="destroy">
      <inputDependencies>
        <dependency source="getResultsTask" sourceMessageType="output"/>
      </inputDependencies>
    </taskDefinition>
  </subProcesses>
  <outputDependencies>
    <dataDependency source="createTask" sourceMessageType="fault"
      sourceMessageName="SoaplabException" sinkMessageType="fault"
      sinkMessageName="SoaplabException"/>
    <dataDependency source="runTask" sourceMessageType="fault"
      sourceMessageName="SoaplabException" sinkMessageType="fault"
      sinkMessageName="SoaplabException"/>
    <dataDependency source="waitForTask" sourceMessageType="fault"
      sourceMessageName="SoaplabException" sinkMessageType="fault"
      sinkMessageName="SoaplabException"/>
    <dataDependency source="getResultsTask" sourceMessageType="fault"
      sourceMessageName="SoaplabException" sinkMessageType="fault"
      sinkMessageName="SoaplabException"/>
    <dataDependency source="destroyTask" sourceMessageType="fault"
      sourceMessageName="SoaplabExcepion" sinkMessageType="fault"
      sinkMessageName="SoaplabException"/>
    <dependency source="destroyTask" sourceMessageType="output"
      sinkMessageType="output"/>
    <dataDependency source="getResultsTask" sourceMessageType="output"
      sinkMessageType="output"/>
  </outputDependencies>
</processDefinition>
```

The π-calculus mapping for this example is:

```
ti(TalismanRequest).ci<CreateJobRequest> |
ci(CreateJobRequest).createTask.(ri<RunRequest> + f<SoaplabException>) |
ri(RunRequest).runTask.(wi<waitForRequest> + f<SoaplabException>) |
wi(WaitForRequest).waitForTask.(gi<GetResultsRequest> + f<SoaplabException>) |
gi(GetResultsRequest).getResultsTask.((go<getResultsResponse> | di<DestroyRequest>) +
f<SoaplabException>) |
di(DestroyRequest).destroyTask.(do<DestroyResponse> + f<SoaplabException>) |
(go(GetResultsResponse) | do(DestroyResponse)).to<TalisamnResponse> |
f(SoaplabException).tf<SoaplabException>
```

5.2 Process Order Application

This application involves the processing of a customer's order. It has been modelled as a process *processOrderApplication* which contains four constituent tasks: *paymentAuthorisation*, *checkStock*, *dispatch* and *paymentCapture*. The relationship between the tasks is shown below. To process an order, *paymentAuthorisation* and *checkStock* tasks are executed concurrently. If both complete successfully then *dispatch* task is started and if that task is successful the *paymentCapture* task is started. The

internal structure of a process can be modified without affecting the tasks which supply it with inputs or use it for inputs. In this case it would be possible to change the payment and stock management policies, for example, causing payment capture even if the item is not presently in stock (a regrettable practice), or the addition of a task which could check the stock levels of the suppliers of the company, and arrange direct dispatch from them.

The wsdl:portType for the application is shown below:

```
<wsdl:portType name="purchaseOrderApplicationPT">
  <wsdl:operation name="paymentAuthorisation">
    <wsdl:input message="paymentAuthorisationRequest"
      name="paymentAuthorisationRequest"/>
    <wsdl:output message="paymentAuthorisationResponse"
      name="paymentAuthorisationResponse"/>
  </wsdl:operation>
  <wsdl:operation name="checkStock">
    <wsdl:input message="checkStockRequest" name="checkStockRequest"/>
    <wsdl:output message="checkStockResponse" name="checkStockResponse"/>
  </wsdl:operation>
  <wsdl:operation name="dispatch">
    <wsdl:input message="dispatchRequest" name="dispatchRequest"/>
    <wsdl:output message="dispatchResponse" name="dispatchResponse"/>
  </wsdl:operation>
  <wsdl:operation name="paymentCapture">
    <wsdl:input message="paymentCaptureRequest" name="paymentCaptureRequest"/>
    <wsdl:output message="paymentCaptureResponse" name="paymentCaptureResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

The process composition is shown below:

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

<processDefinition name="processOrderApplicationProcess"
portType="ws:processOrderApplicationPT" operation="processOrderApplication"
logicalHost="ws:server1" xmlns:ws="http://myservice.com/processOrderApplication.wsdl"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <subProcesses>
    <taskDefinition name="paymentAuthorisationTask"
      portType="ws:processOrderApplicationPT" operation="paymentAuthorisation"
      logicalHost="ws:server1" >
      <inputDependencies>
        <dataDependency source="processOrderApplicationProcess"
          sourceMessageType="input"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="checkStockTask" portType="processOrderApplicationPT"
      operation="checkStock"  logicalHost="ws:server1" >
      <inputDependencies>
        <dataDependency source="processOrderApplicationProcess"
          sourceMessageType="input"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="dispatchTask" portType="processOrderApplicationPT"
      operation="dispatch"  logicalHost="ws:server1" >
      <inputDependencies>
        <dependency source="paymentAuthorisationTask" sourceMessageType="output"/>
        <dataDependency source="checkStockTask" sourceMessageType="output"/>
      </inputDependencies>
    </taskDefinition>
    <taskDefinition name="paymentCaptureTask" portType="processOrderApplicationPT"
      operation="paymentCapture"  logicalHost="ws:server1" >
      <inputDependencies>
        <dependency source="dispatchTask" sourceMessageType="output"/>
```

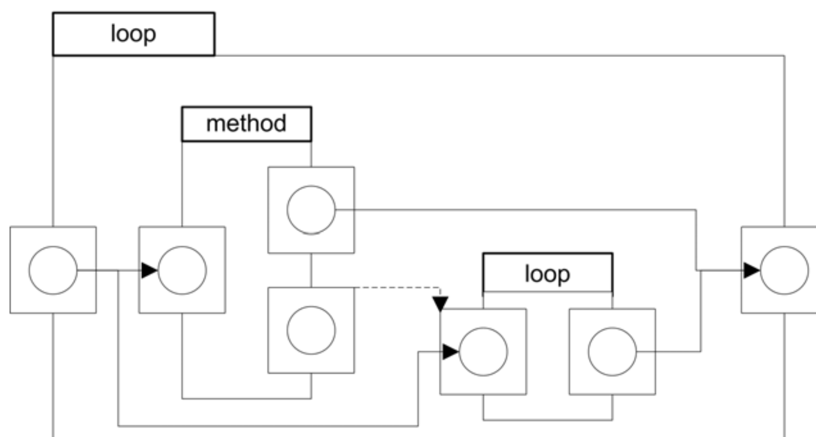
```
        <dataDependency source="paymentAuthorisationTask" sourceMessageType="output"/>
      </inputDependencies>
    </taskDefinition>
  </subProcesses>
  <outputDependencies>
    <dependency source="paymentAuthorisationTask" sourceMessageType="fault"
      sourceMessageName="paymentAuthorisationFault" sinkMessageType="fault"
      sinkMessageName="processOrderApplicationFault"/>
    <dependency source="checkStockTask" sourceMessageType="fault"
      sourceMessageName="checkStockFault" sinkMessageType="fault"
      sinkMessageName="processOrderApplicationFault"/>
    <dependency source="dispatchTask" sourceMessageType="fault"
      sourceMessageName="dispatchFault" sinkMessageType="fault"
      sinkMessageName="processOrderApplicationFault"/>
    <dependency source="paymentCaptureTask" sourceMessageType="output"
      sinkMessageType="output"/>
    <dataDependency source="dispatchTask" sourceMessageType="output"
      sinkMessageType="output"/>
  </outputDependencies>
</processDefinition>
```

The π-calculus mapping for this example is:

```
poi(ProcessOrderRequest).(pai<PaymentAuthorisationRequest> | csi<CheckSockRequest>) |
pai(PaymentAuthorisationRequest).paymentAuthorisationTask.(pci<PaymentCaptureRequest1> |
di<DispatchRequest1>) |
csi(CheckStockRequest).checkStockTask.(di<DispatchRequest2> + csf<CheckStockFault>) |
(di(DispatchRequest1) | di(DispatchRequest2).dispatchTask.((pci<PaymentCaptureRequest2>
|  do<DispatchResponse>) + df<DispatchFault>) |
(pci(PaymentCaptureRequest1) |
pci(PaymentCaptureRequest2)).paymentCaptureTask.pco<PaymentCaptureResponse> |
(pco(PaymentCaptureResponse) | do(DispatchResponse)).poo<ProcessOrderResponse> |
(csf(CheckStockFault) + df(DispatchFault)).pof<ProcessOrderFault>
```

5.3 Looping example

This example shows how looping can be achieved. The figure below illustrates the process structure associated with this example, it shows a task that is executed until it completes successfully. If the task produces an output message then the process terminates, if the task produces a fault message then the process re-executes the task.



The wsdl:portType would be:

```
<wsdl:portType name="Generic">
  <wsdl:operation name="method">
    <wsdl:input message="MethodRequest" name="MethodRequest"/>
    <wsdl:output message="MethodResponse" name="MethodResponse"/>
```

```
    <wsdl:fault message="MethodException"/>
  </wsdl:operation>
</wsdl:portType>
```

The process composition would be:

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

<processDefinition name="LoopingProcess" portType="ws:loopingProcessPT" operation="loop"
  lateInstantiation="true" logicalHost="ws:server1"
  xmlns:ws="http://myservice.com/Generic.wsdl"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <subProcesses>
    <taskDefinition name="genericTask" portType="ws:Generic" operation="method"
      logicalHost="ws:server1" >
      <inputDependencies>
        <dataDependency source="LoopingProcess" sourceMessageType="input"/>
      </inputDependencies>
    </taskDefinition>
    <processDefinition name="loopTask" definition="LoopingProcess">
      <inputDependencies>
        <dependency source="genericTask" sourceMessageType="fault"/>
        <dataDependency source="LoopingProcess" sourceMessageType="input"/>
      <inputDependencies>
    </processDefinition>
  </subProcesses>
  <outputDependencies>
    <dataDependency source="genericTask" sourceMessageType="output"
      sinkMessageType="output"/>
    <dataDependency source="loopTask" sourceMessageType="output"
      sinkMessageType="output"/>
  </outputDependecies>
</processDefinition>
```

The π-calculus mapping for this example is:

```
!(li(LoopingProcessRequest).gi<MethodRequest> |
gi<MethodRequest>.genericTask.(go<MethodResponse> + li<LoopingProcessRequest>) |
go(MethodResponse).lo<LoopingProcessResponse>)
```

6 Schema

```
<?xml version="1.0"?>

<xsd:schema targetNamespace="http://schemas.adapt.org/process-definition-2.1/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://schemas.adapt.org/process-definition-2.1/"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/" schemaLocation="wsdl.xsd"/>
  <xsd:element name="processDefinition" type="tns:ProcessDefinitionRootType"/>
  <xsd:complexType name="ImportType">
    <xsd:attribute name="namespace" type="xsd:anyURI" use="required"/>
    <xsd:attribute name="location" type="xsd:anyURI" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="ProcessDefinitionRootType">
    <xsd:sequence>
      <xsd:element name="import" type="tns:ImportType" minOccurs="0"
        maxOccurs="unbounded"/>
      <xsd:element name="subProcesses" type="tns:SubProcessesType" minOccurs="0"/>
      <xsd:element name="outputDependencies" type="tns:DependencyListType"
        minOccurs="0"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="tns:ActionAttributes"/>
    <xsd:attribute name="targetNamespace" type="xsd:anyURI"/>
  </xsd:complexType>
  <!-- Elements to describe the process and its structure -->
  <xsd:complexType name="ProcessDefinitionType">
```

```
    <xsd:choice>
      <xsd:sequence>
        <xsd:element name="inputDependencies" type="tns:DependencyListType"
          minOccurs="0"/>
        <xsd:element name="subProcesses" type="tns:SubProcessesType" minOccurs="0"/>
        <xsd:element name="outputDependencies" type="tns:DependencyListType"
          minOccurs="0"/>
      </xsd:sequence>
    </xsd:choice>
    <xsd:attributeGroup ref="tns:ActionAttributes"/>
    <xsd:attribute name="definition" type="xsd:QName"/>
  </xsd:complexType>
  <xsd:complexType name="TaskDefinitionType">
    <xsd:choice>
      <xsd:element name="inputDependencies" type="tns:DependencyListType"
        minOccurs="0"/>
    </xsd:choice>
    <xsd:attributeGroup ref="tns:ActionAttributes"/>
    <xsd:attribute name="port" type="xsd:QName"/>
  </xsd:complexType>
  <xsd:complexType name="SubProcessesType">
    <xsd:sequence>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="taskDefinition" type="tns:TaskDefinitionType"/>
        <xsd:element name="processDefinition" type="tns:ProcessDefinitionType"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Elements to describe the dependencies between tasks and processes -->
  <xsd:complexType name="DependencyType">
    <xsd:attributeGroup ref="tns:DependencyAttributes"/>
  </xsd:complexType>
  <xsd:complexType name="DataDependencyType">
    <xsd:attributeGroup ref="tns:DependencyAttributes"/>
    <xsd:attribute name="sourcePartName" type="xsd:string" use="optional"/>
    <xsd:attribute name="sinkPartName" type="xsd:string" use="optional"/>
  </xsd:complexType>
  <xsd:complexType name="DependencyListType">
    <xsd:sequence>
      <xsd:element name="dependency" type="tns:DependencyType" minOccurs="0"
        maxOccurs="unbounded"/>
      <xsd:element name="dataDependency" type="tns:DataDependencyType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Utility definitions to define common attributes of tasks/processes and
    dependencies -->
  <xsd:attributeGroup name="ActionAttributes">
    <xsd:attribute name="name" type="xsd:QName" use="required"/>
    <xsd:attribute name="portType" type="xsd:QName" use="optional"/>
    <xsd:attribute name="operation" type="xsd:NCName" use="optional"/>
    <xsd:attribute name="lateInstantiation" type="xsd:boolean" use="optional"
      default="false"/>
    <xsd:attribute name="logicalHost" type="xsd:QName" use="required"/>
  </xsd:attributeGroup>
  <xsd:attributeGroup name="DependencyAttributes">
    <xsd:attribute name="source" type="xsd:QName" use="required"/>
    <xsd:attribute name="sourceMessageType" type="tns:MessageTypeType" use="required"/>
    <xsd:attribute name="sourceMessageName" type="xsd:string" use="optional"/>
    <xsd:attribute name="sinkMessageType" type="tns:MessageTypeType" use="optional"/>
    <xsd:attribute name="sinkMessageName" type="xsd:string" use="optional"/>
    <xsd:attribute name="priority" type="xsd:integer" use="optional" default="0"/>
  </xsd:attributeGroup>
  <xsd:simpleType name="MessageTypeType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="input"/>
      <xsd:enumeration value="output"/>
      <xsd:enumeration value="fault"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

7 References

[1] Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/SOAP/ as viewed September 2003
[2] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl.html as viewed September 2003
[3] UDDI Specification, OASIS http://uddi.org/ as viewed September 2003
[4] Architectural Support for Dynamic Reconfiguration of Distributed Workflow Applications, Shrivastava, S.K. and Wheater, S.M. IEE Proceedings - Software, Volume 145, Issue 5, pp 155-162 Institution of Electrical Engineers (IEE), ISSN: 1462-5970, 1998
[5] R.Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, 1999
[6] Deliverable 6: Service Specification, Adapt Project Deliverable, September 2003

Appendix A: A System for Distributed Enactment of Composite Web Services

A System for Distributed Enactment of Composite Web Services

S.J.Woodman[1], D.J.Palmer[1], S.K.Shrivastava[1], S.M.Wheater[2]

[1]School of Computing Science, University of Newcastle, Newcastle upon Tyne, UK
{S.J.Woodman, Doug.Palmer, Santosh.Shrivastava}@ncl.ac.uk
[2]Arjuna Technologies Limited, Nanotechnology Centre, Newcastle upon Tyne, NE1 7RU, UK
Stuart.Wheater@arjuna.com

Abstract. Availability of a wide variety of Web services over the Internet offers opportunities of providing new services built by composing them out of existing ones. Service composition poses a number of challenges. A composite service can be very complex in structure, containing many temporal and data-flow dependencies between their constituent services. However, constituent service operations must be scheduled to run respecting these dependencies, despite the possibility of intervening processor and network failures. The architecture must be scalable, providing a decentralised coordination of service execution rather than based on a centralised scheduler; this is particularly important for services spanning different organisations, where reliance on centralised coordination would be impractical. This paper presents the design and implementation of DECS: a workflow management system for Distributed Enactment of Composite Services. A novel feature of DECS is the separation between specification of service composition and its enactment. A DECS service specification can be deployed either for centralised or decentralised coordination, depending upon inter-organisational requirements. A prototype implementation of DECS has been performed using J2EE middleware. The paper describes the DECS task model for specifying service composition and the middleware services that have been implemented in J2EE for coordinating service execution.

1. Introduction

It is becoming increasingly common for a Web service to make use of Web services offered by different organisations. We term such an inter-organisational service a "Composite Web Services" (CSs). There is much research interest in developing high-level tools for CS creation and management, including service specification languages and run time environments for coordinating the execution of (enactment of) constituent services. This paper presents the design and implementation of DECS: a workflow management system for Distributed Enactment of Composite Services. There are several features DECS that make it novel and distinct from existing workflow management systems for Web service enactment.

(i) *Flexible Coordination*: workflow management systems for Web service enactment specify the composition of a CS as a business process. Such a specification should be sufficiently abstract, uncluttered with specific details of enactment. Specifying the enactment of the business process is a concrete operation which requires further information that is critically dependent on organisational issues. For example, the organisations involved in a CS may have a peer-to-peer business relationship, in which case, a decentralised enactment seems a natural choice, with each organisation responsible for its part of the process. Whereas in a hierarchic relationship, a centralised enactment may well be deemed more appropriate. DECS supports the separation between the specification of service composition and its enactment, enabling the organisations deploying the service to decide how they wish to coordinate it, rather than the designer of the business process.

DECS provides the option of both centralised coordination as shown in Figure 1 and decentralised coordination, as shown in Figure 2. Should the coordination of the service be spread across multiple servers as in Figure 2 a higher level of fault tolerance is provided. In such cases, each server makes the invocations of the constituent web services for its part of the CS, and communicates via a coordination protocol with its peers to orchestrate the overall execution. Should a coordinating server fail or leave, it is possible to move the CSs which that server was coordinating to another server. The cost of doing so is proportionate to the number of CSs being coordinated and the complexity of those CSs.

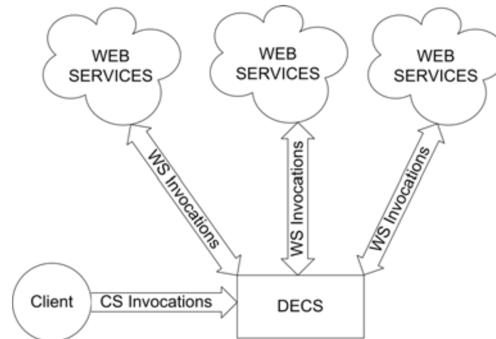


Figure 1 Centralised Coordination of a Composite Service

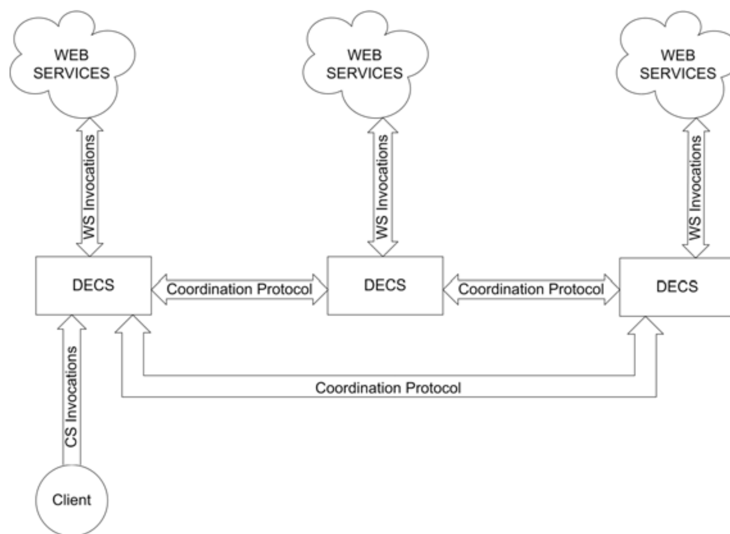


Figure 2 Decentralised Coordination of a Composite Service

(ii) *Support for reconfiguration*: It is expected that the execution of a CS could take a long time to complete, of the order of days or weeks, and may contain long periods of inactivity often due to the constituent applications requiring user inputs. It should be possible therefore to reconfigure a CS dynamically because, for example, services may be moved, machines may fail or user's requirements may change. DECS provides basic system support for reconfiguration (see section 2).

(iii) *Support for preserving organisational Autonomy*: A further advantage of distributed coordination is that it enables organisations to collaborate whilst maintaining autonomy. When an inter-organisation business process is created, it can be deployed such that each organisation coordinates the parts which they are responsible for and which act upon their internal data. Each organisation is aware of the data relating to

those tasks which it is coordinating, and also certain pieces of data which they have requested from the other organisations.

The remainder of this paper is structured as follows: section 2 gives an overview of the system; section 3 describes the task model of DECS; in section 4 we describe the system architecture and its implementation; section 5 we discuss patterns for distributing coordination; section 6 describes related work, finally section 7 concludes the paper.

2. System Overview

We discuss how our system has been designed to meet the application requirements implied above, namely: scalability, dependability, interoperability, dynamic reconfiguration and flexible task composition. The design of the system has been influenced by our earlier work on the OPENflow distributed workflow management system [1,2].

- Scalability: Once a composite service is deployed, there is no reliance on any centralised service which could limit scalability. The decoupling of the business process definition from the deployment aids scalability too – it is possible to reconfigure the business process at run-time if more resources are needed.
- Flexible Task Composition: The system supports a simple yet powerful task model (see section 3). This allows the composition of complex services from simple services located both on the local machine and remote web services. A task can perform application specific input selection (e.g. obtain input from one of several sources) and terminate in one of several outcomes.
- Dependability: Dependability is implemented at two levels in DECS: application level and system level. Due to the flexible task composition model mentioned above it is possible to specify different tasks and processes to handle a variety of exceptional circumstances, for instance, compensating tasks, alternative tasks etc. At the system level, DECS makes use of the facilities provided by modern component middleware (J2EE) to ensure that all information relating to the composite service: its structure, data and state are persistent. All interactions between different modules of DECS make use of transactions to ensure that the data remains consistent and tasks do not interfere with each other. If a coordinating node fails mid process, it is possible to recreate the state of the CS and continue processing from where the failure occurred. This is discussed further in Section 4.6.
- Interoperability: The system has been designed to run in any J2EE compliant application server, thereby supporting system level interoperability. As DECS can invoke web services with arbitrary interfaces, application level interoperability is provided.
- Dynamic reconfiguration: The task model referred to earlier is expressive enough to represent dependencies (dataflow and notification) between tasks. The execution environment exposes low level operations for making changes to the structure of the CS by altering the tasks and the dependencies between them. We are currently working on making this dynamic reconfiguration support transactional to ensure that changes are carried out in a consistent manner despite concurrent reconfigurations and machine failures. This work is being done along the lines of our earlier implementation [2].

3. Task Model

DECS makes use of a flexible task model to describe the structure of Composite Services. The schema for such a task model must be expressive enough to allow arbitrary CSs to be defined. Our schema is defined in terms of tasks, temporal dependencies and data dependencies. A task represents an application specific unit of work that requires specified input data and produces specified output data and corresponds to an invocation of a web service. A task instance is modelled as receiving one input message, and sending multiple output messages. The web service invoked is treated as a black box with the input and output data specified by the WSDL document associated with the service [3].
A temporal dependency represents the situation where a down-stream task cannot start until an up-stream task has terminated in a particular state. For example, a goods dispatch task should not be invoked until a payment-capture task has completed successfully. A data dependency indicates that a down-stream task requires input data from the up-stream task. Each data dependency implicitly has a temporal dependency associated with it. For example, a dispatch task requires shipping-address which is part of the output from order task. Implicitly this is not available until the order task has completed [1].

Some of the salient points of the task model which aid flexible composition of CSs are presented below:

- Alternative input sources: Any part of the input data can be acquired from more than one source. This enables the introduction of redundant data sources providing application level fault tolerance. The input data for a task is described at the granularity level of wsdl:part. Support for finer granularity data than wsdl:part is described in Section 7.
- Alternative outputs: A task can terminate in one of several states producing distinct outcomes. In terms of WSDL, one of these states will correspond to an "output" and all others to a "fault". This allows different down-stream tasks to be executed depending on the outcome of an up-stream task. For example, compensation in the event of a fault.
- Compound Tasks: A task can be composed from other tasks. The composed tasks themselves can be simple tasks or compound tasks. This is the principle way of providing abstraction.
- Genesis tasks: A genesis task represents a placeholder for a task structure and is used for on-demand instantiation. This allows the system to only instantiate the parts of a complex task which are necessary, and also allows execution of repetitive tasks.

Each input and output message can contain several parts, each representing a piece of data which can be requested by other tasks. Task t_2 in Figure 3 has one input message containing two parts, namely i_1 and i_2. There are two output messages for task t_2, O_1 and O_2, each containing one part, o_1 and o_2 respectively. A task begins its life in the wait state, awaiting its input data to be complete. When the input message is complete (i.e., all the data dependency and temporal dependencies are satisfied) the task enters the ready state. If alternative inputs become available simultaneously the one with the highest priority is used. Note that the source of an input part can be from an output message (e.g. d_1), or an input message (e.g. d_2), the latter represents the case when an input is consumed by more than one task. Temporal dependencies are depicted as a dotted line, for example n_1 and data dependencies are shown as solid lines.

A compound task such as t_1 can contain multiple output messages, with each part having several possible sources. The task will terminate when one output message is complete. If multiple output messages become available simultaneously, the one with the highest priority will be chosen.
In section 6 on related work, we compare our task model with the recent proposal for Web service enactment, BPEL [4].

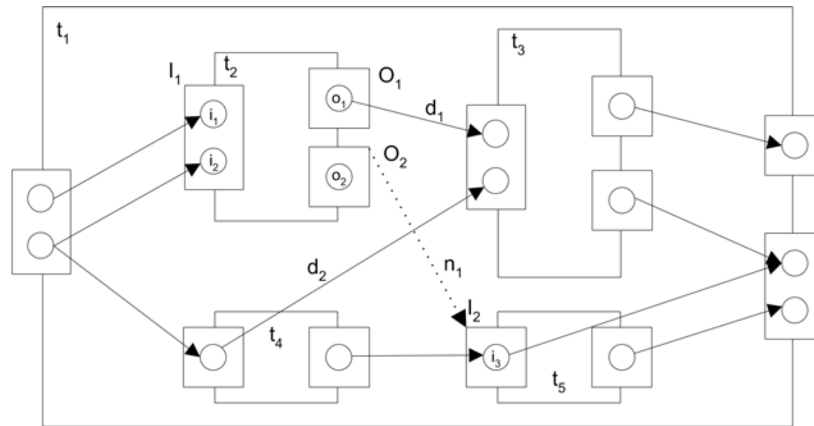


Figure 3 A Compound Process

4. System Design and Implementation

The DECS system has been designed and implemented using J2EE middleware. The current version has been tested to run within the JBoss application server [5].

4.1. Overall structure

The structure of the system is shown in Figure 4. The figure is intended to show how a client's request to execute CS enters the Process Initiator; an instance of the process definition is taken from the Process Definition Repository (PDR) and added to the Process Instance Repository (PIR). The coordinator then uses this data to invoke the web services which compose the CS and inform other coordinating servers of data in which they have registered an interest. Every client request instantiates a new and unique process definition instance. The J2EE technologies used to implement the different modules are shown in Figure 5 which matches the structure of Figure 4.
Using the J2EE environment allows DECS to make use of the rich set of functionality which J2EE application servers provide. For example, uniform access to persistent storage, flexible transaction control, a unified security model. DECS utilises the Entity Beans, Session Beans and Message Driven Beans from the J2EE architecture. An Entity Bean represents a business object that exists in the enterprise application. The application server controls access to Entity Beans, guaranteeing atomicity, consistency, isolation and durability. A Session Bean provides the business logic for a J2EE application. Clients call methods of the Session Bean to interact with the application. Message Driven Beans provide asynchronous behaviour by acting as message listeners for the Java Message Service (JMS). JMS facilitates the exchange of messages among software applications over a network. The prototype of DECS is interoperable in that it will run in any J2EE application server as no proprietary extensions have been exploited.

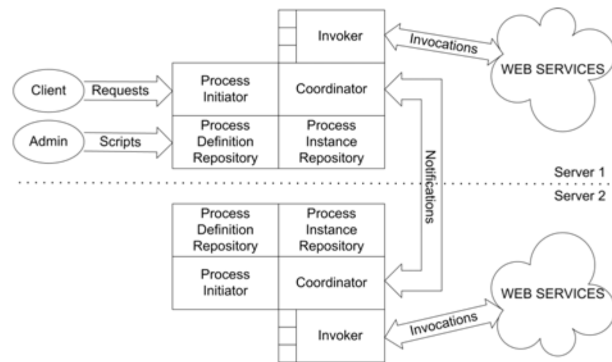


Figure 4 System Architecture

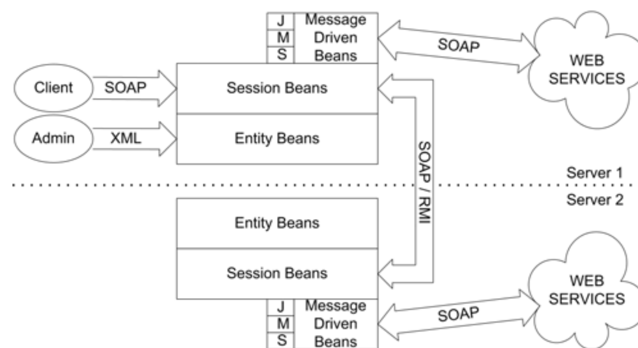


Figure 5 System Technologies

4.2 Process Initiator

The process initiator is able to dynamically deploy web service endpoints which can be used by a client to invoke the web service which corresponds to a process. When a composite service is deployed in the system, an endpoint is generated which allows a client to invoke it. This enables us to achieve transparency from the client's perspective, whereby the client may be unaware that they are invoking a web service which is implemented as a process. This style differs from many coordination engines which expose one interface to invoke many processes. For example, web service clients of the CARNOT workflow engine must send a message to the "WsWorkflowSessionService" endpoint passing the name of the process to invoke and a context. One of the problems with this style of interaction is that there is no standard way of locating the required service. Processes which can be invoked using the Process Initiator can be advertised using any current advertising method, e.g. UDDI [6].

When designing a CS, one server is designated as the root controller for that CS. This is usually the server which will coordinate the first task in the CS but this is not a requirement. The root controller is the only server on which the Process Initiator deploys an endpoint. When a CS is invoked by a client, an instance of the process definition is created in the PIR based on the process definition stored in the PDR. This contains all the data necessary to run the CS, such as the tasks involved and the inter-task dependencies which must be satisfied. The root controller also sends a message to the other servers coordinating the execution requesting that they create an instance of the fragment of the process definition that they are responsible for. This is discussed further in Section 5. The initial values received in the client request are then added to the "root" task in the repository and those tasks whose input dependencies are satisfied are invoked.

4.3 Process Definition Repository (PDR)

The Process Definition Repository stores process definitions and provides a method for instantiating an instance of a process. DECS provides tool support adding an XML based process definition script to the PDR. This can either be a complete process definition for centralised coordination or a partial definition if decentralised coordination is required. The PDR is implemented as a collection of Java Entity Beans deployed in the J2EE application server.

A process definition is represented by a schema which matches the task model described in the previous section, in terms of tasks and dependencies. We have made use of the concepts of a scripting language developed for a previous distributed workflow system and adapted it to suit web services style invocations. The scripting language was designed to express composite service composition and inter-task dependencies of fault tolerant distributed applications whose executions could span arbitrary large durations [7].

4.4 Process Instance Repository (PIR)

The system will instantiate a process instance based on a process definition for each request from a client. The data related to each process instance is stored in the PIR which is also implemented as a collection of Entity Beans The state of the process instance is persistent and each change is made persistent, this means that coordination of process instances can be continued after machine failure.

Creating a process instance from a process definition involves three steps:

- Create the local structure of the process instance in the PIR according to the definition stored in the PDR.
- Instantiate the fragments of the schema which reside on remote nodes. This is described further in Section 6.
- Insert the initial values into the process instance from the client request.

4.5 Coordinator

The coordinator in DECS orchestrates the execution of the CS across multiple nodes. This involves checking for input availability, maintaining state of the task and propagating the results both locally and remotely. The prototype uses Session Beans to implement the business logic associated with coordination.

When a process definition is instantiated all the tasks are in the wait state. As input data is added to the tasks they are checked to see if their input message is complete. Once the input message is complete the task moves to a ready state and is put on a persistent JMS queue to indicate that the task is invocable. At this stage, the coordinator checks to see if there are any input dependencies or notifications on the task's input message. If local dependencies (either data or temporal) exist, the data is propagated locally to these tasks. If a dependency is remote, this initiates a notification of the data to the remote coordinator. These are shown in Figure 8(iii) as d_1 and n_1 respectively. When propagating data to the remote coordinator, either SOAP or Java RMI can be used. SOAP is intended to be used as the primary communication method but Java RMI can be used to optimise the communications if both coordinators are located on the same network. In both cases, the local coordinator must communicate with the remote coordinator via an RPC style call. The parameters of the notification include the unique identifier of the process instance which is the source of the data and the data value. The action of inserting the data part to the remote task has the side effect of checking the task to see if it can be executed. The semantics of adding any data, local or remote, to a

task results in the task being executed as soon as its input message is available. When the output message of the root task is complete, the Process Initiator will create a SOAP response and send it back to the client.

4.6 Invoker

The invoker is responsible for invoking the web services which comprise the composite service. The invoker is implemented as Message Driven Beans in the application server. This allows us model the asynchronous behaviour that is required to invoke the constituent web services when their dependencies have been fulfilled.
To cope with the consequences of coordinator failure, the designer of the service is currently able to specify one of two actions to be taken on restart for each task which was executing when the failure occurred. These correspond to whether they wish the constituent web services to be invoked "at most once" or "at least once". If the designer specifies they wish to use "at most once" semantics, transactions are not used in the invoker. The invoker is not aware that an instance of this service was executing when it failed, so it does not attempt to re-invoke the web service request. If "at least once" semantics are required, everything performed by the invoker is within one transaction. This includes obtaining the input data for the web service, invoking that service, receiving the results and adding them to the PIR. If the invoker fails at any point within this transaction, it will be rolled back on restart. This results in the web service being invoked with the same input data again, once the system has recovered. If an end-to-end transaction protocol were available, it would be possible to achieve "exactly once" execution semantics for the constituent web services of the process. This is because any failure in the coordinator would be propagated to the constituent web services which could rollback their execution too. On restart, the invoker would send the SOAP request again and eventually a successful execution would occur.

5. Distribution Patterns

It is the responsibility of the designer of the service to specify how they wish to distribute the coordination of the service. Simple CSs may be coordinated centrally, as shown in Figure 6. This is the simplest scenario, where a central node performs all the invocations of the services necessary to complete the CS. More complex scenarios can be built, where the coordination of the CS is divided between multiple nodes which run DECS. One example is given in **Figure 7** where the service is divided between two nodes. How to divide the service is an application specific decision. If we consider the notion of a Virtual Enterprise where a set of companies wish to collaborate to provide a service, the division of coordination could be along organisational boundaries. This allows each organisation to coordinate their own part of the service, and possibly utilise their private services. For example, company S wishes to sell a product and use company D to deliver the product. The composite service could be divided such that nodes at S coordinate the ordering of the product and payment and then the nodes at D coordinate scheduling of the delivery of the product. A division such as this has some advantages: firstly, S can integrate the order with their own procurement process allowing re-ordering of stock if necessary and D can integrate with their private delivery scheduling services; secondly, data security is higher as only the minimum amount of data is passed across from one organisation to the other to allow the CS to continue its execution.

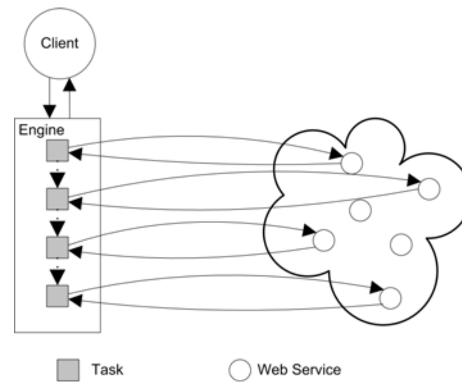


Figure 6 Centralised Coordination

The system aims to give each node the minimum information necessary to coordinate the execution of that part of the CS. Each node only stores the data about the tasks which it is coordinating, the internal dependencies which must be satisfied and the external notifications which must be sent and received. The node is not aware of what the other nodes are doing, or what tasks they are coordinating. This is intended to provide autonomy; it makes it attractive to businesses that do not wish to disclose all of their internals but do wish to integrate their business processes with a trading partner.

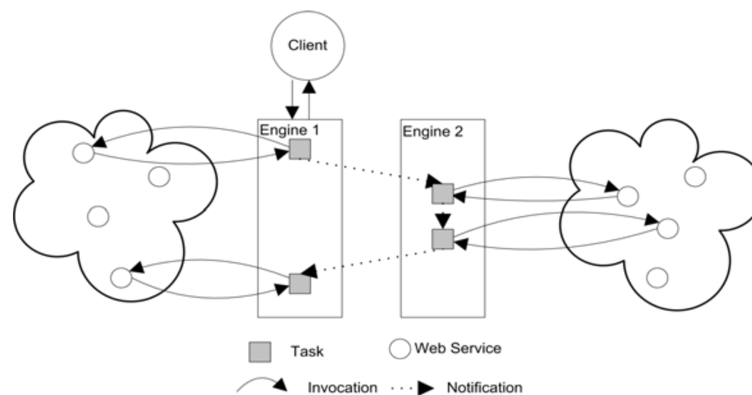


Figure 7 Decentralised Coordination

Figure 8 shows an example of how a very simplistic CS, A could be divided to run over three servers, X, Y and Z. Figure 8(i) shows the overall CS with figures 8(ii) to 8(iv) showing the three server's views of the service. Server X is delegated as the controller of the CS so it is this server which exposes an endpoint allowing the service to be invoked. When a client request is received at X for service A, the first task to be executed will be B. When B completes, there are no dependent tasks at server X, but there is a notification request for the results to be sent to Y. When Y receives the results of B via a notification, task C is executed (by Y) and on completion the results sent via a notification to Z. As there is also a dependency for task E at Y, the data is propagated locally to task E's input message. As the input message for task E is not yet complete, the task is not invoker. At server Z, task D's input message is complete by receiving the notification from Y so task D is executed. A notification is sent from Z to Y on completion of D which causes the input message of task E to be complete and thus task E fired. Completion of task E causes a notification to be sent to X with the results which are used as input for task F. On completion of task F, the output message for the compound task A is complete, so the result is sent to the client.

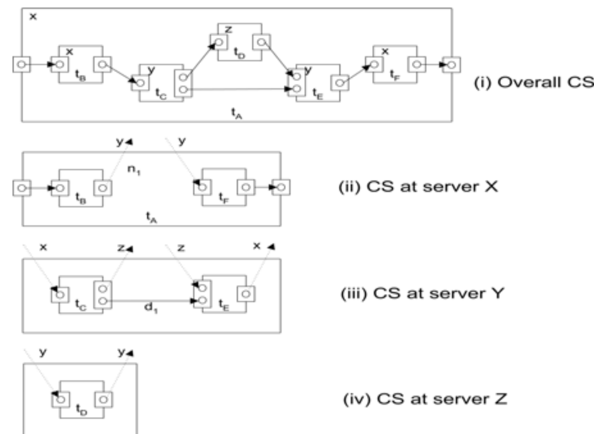


Figure 8 Division of a CS

6. Related Work

There are several industry led efforts aimed at developing (often competing!) standards for specifying, composing and coordinating the execution of CSs. The specifications are still evolving and often ill defined [8]. We compare our task model with a recently proposed Web service execution language standard. The aim of Business Process Execution Language for Web Services (BPEL) [4] is to provide a standard for specifying business process behaviour and business process interactions, for applications composed from Web services. We are going to focus on a comparison of the approach taken by BPEL for specifying business process behaviour and that taken by DECS.

Like DECS the BPEL process model allows business partners interact through peer-level conversations, using both synchronous and asynchronous messages. These conversations are carried out between the partners using specified sets of Web services. BPEL has been designed to allow the coordination of distributed web services in a centralised manner. The coordination itself was not designed to be distributed. The coordination model is tailored towards implementations that have a centralised state upon which a rich set of activities can act. The result is that providing a decentralised enactment engine is difficult especially if the intention is to deploy the workflow in a wide area network. In comparison, the DECS task model was designed such that each task holds a small amount of state and there is a minimal set of services which are able to act on this state. This results in a model which is easier to distribute as each task encapsulates its own state and it is easier to migrate task coordination to another node.

BPEL relies on specifying a large set of explicit control flow activities such as forking, joining and conditionals. Conversely, control flow in DECS is implicit and specified through dependencies. These constructs have been found to be sufficient to specify complex processes. Another consequence of BPEL having such an extensive set of features which can be combined to specify a process is that the resulting process cannot be easily analyzed to verify such properties as eventual termination. The DECS model is much simpler and amenable to analysis, allowing both temporal and correctness properties to be checked [9].

There are a number of systems available which coordinate the orchestration of composite services produced both in the academic community and in industry. Some of these are discussed below and briefly compared and contrasted to DECS.

Collaxa [10] is well-known product that can enact composite services specified in BPEL. It is a centralised coordination engine; this is inevitable, given the observations on BPEL above. As such, it cannot offer the same level of flexibility in deployment scenarios, dependability and scalability that DECS intends to provide.
Both eFlow [11] and BioOpera [12] explore the declarative composition of services but concentrate on a centralised orchestration model. Successful efforts have been made in eFlow to allow dynamic refactoring of services although this is simplified due to a central, global view being available.
DECS has been designed to run in a J2EE environment to allow portability and ease of integration into existing enterprise applications. DySCo [13] offers many similar features to DECS, such as decentralised coordination but portability has not been addressed to the same level: it is not designed to run in standard middleware such as J2EE. Conversely, CARNOT [14] offers portability through a J2EE implementation, but does not support distributed orchestration of composite services.
Another approach which has been explored is that of SELF-SERV [15], based on a declarative state-chart oriented language. It also includes a peer-to-peer coordination model and provides support for equivalent services through the use of service communities. It comes closest to DECS in terms of design aims and functionality.

7. Concluding Remarks

We have presented the design and implementation of DECS: a workflow management system for Distributed Enactment of Composite Services. A novel feature of DECS is the separation between specification of service composition and its enactment. A DECS service specification can be deployed either for centralised or decentralised coordination, depending upon inter-organisational requirements. A prototype implementation of DECS has been performed using J2EE middleware.
A suite of common services is being developed as part of the DECS. Such services include:

- User input service: it is likely that some CSs will require input from users at different parts of the execution. For this reason we are developing a servlet based user interface which will allow users to input parameters to be used in the execution. The data entered may be used to determine the consequent flow of execution or to provide advanced error recovery.
- Send and Receive services: In order to allow asynchronous communications services will be developed which will send or receive a message. Tasks which utilise these services can be added to any CS, thus potentially providing a fully asynchronous CS. We envisage more web services becoming available which require message based communications rather than the RPC style services which are common at present. Such document exchange web services are more versatile and allow easier integration of business processes. However, with the introduction of this style of interaction problems are introduced such as message correlation and temporal issues. These will be investigated further.
- Administrative Services: services will be provided which allow a user (with appropriate permissions) to deploy, remove and dynamically reconfigure process definitions. Care must be taken when refactoring a service which is distributed across multiple nodes to ensure that deadlock is prevented. This could occur in cases where tasks are removed upon which a remote task is awaiting a notification. The service will provide mechanisms to ensure that this situation does not occur.

- Transformation of complex types: At present, the system is able to manipulate the flow of data at the granularity of WSDL parts. However, if a complex type is defined in WSDL the system treats this as a black box and cannot address internal fields. We see this as a deficiency in the system so aim to provide a service which is able to transform complex types so that they conform to another schema. This is likely to be done using XSLT, with the designer of the CS providing a style sheet describing the transformation required. An example where this would be useful would be extracting the invoice-number from an invoice type that was returned from the order service and use it as the input to another service.

Acknowledgements

Discussions with Gustavo Alonso clarified our ideas. This work is part-funded by the UK EPSRC under grant GR/N35953/01: "Information Co-ordination and Sharing in Virtual Environments"; by the European Union under Project IST-2001-37126: "ADAPT (Middleware Technologies for Adaptive and Composable Distributed Components"; and by the UK DTI e-Science programme under project "GridMist".

References

[1] S. M. Wheater, S. K. Shrivastava and F. Ranno "A CORBA Compliant Transactional Workflow System for Internet Applications", Proc. Of IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware 98, (N. Davies, K. Raymond, J. Seitz, eds.), Springer-Verlag, London, 1998, ISBN 1-85233-088-0, pp. 3-18.

[2] J. J. Halliday, S. K. Shrivastava and S. M. Wheater, "Flexible Workflow Management in the OPENflow system", Proc. of 5th IEEE/OMG International Enterprise Distributed Object Computing Conference (EDOC 2001), September 2001, Seattle, pp. 82-92.

[3] Web Services Description Language (WSDL) 1.1.http://www.w3.org/TR/wsdl.html as viewed July 2003

[4] Business Process Execution Language for Web Services (BPEL4WS) version 1.1. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/ as viewed July 2003

[5] JBoss application server: www.jboss.org

[6] UDDI Specification, OASIS http://uddi.org/

[7] Ranno, F., Shrivastava, S.K., and Wheater, S.M., "A Language for Specifying the Composition of Reliable Distributed Applications", 18th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS'98, Amsterdam, May 1998, pp. 534-543.

[8] W.M.P. van der Aalst, "Don't go with the flow: Web services composition standards exposed", IEEE Intelligent Systems, Jan/Feb 2003.

[9] C. Karamanolis, D. Giannakopoulou, J. Magee and S.M. Wheater, "Model Checking of Workflow Schemas", Proc. of 4th IEEE/OMG International Enterprise Distributed Object Computing Conference (EDOC 2000), September 2000, Makuhari, Japan.

[10] Collaxa: BPEL Orchestration Engine. http://www.collaxa.com as viewed July 2003

[11] Casati, F., Ilnicki, S., Jin, L., Shan, M., "An Open, Flexible, and Configurable System for E-Service Composition", Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems. 2000. Milpitas, California.

[12] Bausch, W., Pautasso, C., Schaeppi, R., and Alonso, G, "BioOpera: Cluster-aware Computting", 4th IEEE International Conference on Cluster Computing. Chicago, USA.

[13] Piccinelli, G., Finkelstein, A., Williams, S.L., "Service-Oriented Workflows: The DySCO Framework", Proceedings of Euromicro Conference, Antalya, Turkey, 2003

[14] CARNOT Workflow Engine. http://www.carnot.ag/en/ as viewed July 2003

[15] Benatallah, B., Sheng, Q.Z., and Dumas, M., "The Self-Serv Environment for Web Services Composition", IEEE Internet Computing, 2003. 7(1): p. 40-48.