ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Revised Evaluation Plan

Deliverable Identifier:	D16
Delivery Date:	19th August 2004
Classification:	Public Circulation
Authors:	Stuart Wheater
Document version:	4.0 3rd August 2004

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)

Table of Contents

1 ADAPT Evaluation Plan

The purpose of this document is to set out a plan for evaluating the results of the ADAPT project. This evaluation plan will be used to not only to evaluate the project but also to serve as a guide during the project. The resulting evaluation is intended to cover both the functional and non-functional capabilities of the project's results.

The overall strategy for evaluating the non-functional capabilities of ADAPT's results is based around constructing applications that can be used to examine the Availability, Adaptability, Scalability and Performance of the ADAPT platform. To ensure that the evaluation is comprehensive the applications will encompass both basic and composite service support. The details of this strategy are given in section 1.1.

The overall strategy for evaluating the functional capabilities of ADAPT's results is based around two complementary components. Firstly, to provide a proof-of-concept implementation of an application on top of the ADAPT platform, this will be used to provide insight into any inadequacies in the functional capabilities of the ADAPT platform. And secondly, to identify the key high-level technical goals of the project, then based on these goals, to compare the technical capabilities and attributes of the ADAPT approach with those set of existing technologies. The details of this strategy are given in section 1.2.

1.1 Strategy for Evaluating Non-Functional Capabilities and Attributes

The main non-functional goals of the ADAPT project are in the areas of Availability, Adaptability, Scalability and Performance. To make possible the evaluation of these goals a series of applications will be constructed from which quantitative results can be obtained. These applications will be designed to exercise both the Basic and Composite service support of the ADAPT platform. The follow two sections will describe the experiments that will be performed by the applications, on the Basic and Composite service support.

1.1.1 Basic Service Support Experiments

The experiments targeted at the Basic Services support will examine the characteristics of the system via each of the three tiers: web service support (front tier), EJB container support (middle tier) and database container support (back tier). Each of the experiments on a particular tier will naturally also exercise the subsequent tiers, for example, experiments on the middle tier will also exercise the back tier. The experiments that will be performed on the system are described below.

1.1.1.1 Via the Front Tiers

The experiments performed on the system via the front tier will be based around the implementation of one of the WS-I's Sample Application services [34][35] and the trace data used for each experiment will be based on one or more of the WS-I's Sample Application use cases. This system will be used to perform the following experiments:

- The performance of the front tier will be determined by measuring the systems average throughput and average latency. This experiment will be performed with different values of the following parameters:

  - Degrees of replication.
  - Number of simulated clients (scalability).
  - Number of failed replicas (availability).
- The adaptability of the front tier will be determined by measuring the average time it takes to react to a failure (detect and reconfigure) and to react to a recovery. This experiment will be performed with different degrees of replication.

1.1.1.2 Via the Middle Tier

The experiments performed on the system via the middle tier will be based around the existing ECPerf [37] benchmark. ECPerf is a benchmark designed to measure performance and scalability. In addition some experiments will be performed using the implementation of the Enterprise Java Beans that support a WS-I's Sample Application service. Trace data used for each experiment will be derived from one of the WS-I's Sample Application use cases. To make fine-granularity analysis, some micro benchmarks will analyze the behavior for individual components and access patterns. The following experiments will be performed with one or more of these applications:

- Evaluation of the system during normal processing (no joins/leaves the group). These results will be analysed under various system conditions determined by the following parameters:
  - Replication algorithm.
  - Number of servers.
  - Load submitted to the system.
  - Type of application (ECPerf, WS-I, micro-benchmarks)
- Profiling: Analyse of where most of the time is spent: application logic, communication/state-transfer, etc.
- Evaluation of the system after a server crash that causes a failover (server leaves the group). When the failover occurs we will analyse the following behaviour:
  - Semantics observed by the client.
    - Percentage of aborts (current transaction aborts) if applicable.
    - Delay observed by the client from its last request to old server to getting the first response from the new server.
  - Time from crash of old server to time new server is partially or fully operational.
- Evaluation of the system at recovery (crashed or new server joins the group).
  - Time it takes for new server to request join to be fully operational member of the group.
  - Effect of join to the performance in the rest of the system.
    - Throughput/response time observed by the client during recovery time.

1.1.1.3 Via the Back Tier

The experiments performed on the system via the back tier will be based around the specially designed SQL queries and workload patterns that will exercise different characteristics of the database tier [36]. In addition, SQL queries derived form those produced by EJBs that implement a WS-I's Sample Application service will be used for further experiment. This system will be used to perform the following experiments:

- The scalability of configurations with an increasing number of replicas will be quantified in terms of throughput relative to the throughput of a single node. Different workloads ranging from read-only to write-only will be used. The response time will also be measured for increasing loads to find out whether the throughput is increased without increasing significantly the response time (see the technical report in the annex for the experiments run for this evaluation).
- In order to quantify the benefits of online recovery, the throughput and the response time during recovery will be compared for an online recovery and offline recovery. Experiments will be run for increasing log sizes.
- Admission control is in charge of regulating the degree of concurrency within the database by changing the size of the DB connection pool. Experiments will be run to find out:
  - The optimal degree of concurrency for different workloads (CPU-bound, IO-bound).
  - Performance with different fixed size DB connection pools and adaptive size DB connection pools.
- The scalability under different degrees of load imbalance (from totally balanced to totally imbalanced) will be quantified to compare the performance of the middleware with and without load balancing.

In addition to the experiments above some existing benchmarks, such as TPC Benchmark W (TPC-W) [38] will be examined, to see if they can provide any useful addition quantitative results.

1.1.2 Composite Service Support Experiments

The experiments targeted at the Composite Services support will examine the characteristics of the two main software components: the composite service execution engine and the advanced transaction support. The experiments that will be performed on each components of the system are described below.

1.1.2.1 Composite service execution engine

The performance, scalability and adaptability of the composite service execution engine will be determined by performing a series of experiments. These experiments will use a set of process definitions that will exercise differing characteristics of the engine. The process definitions what will be used are:

- "A long chain" of serial executed empty (null operation) tasks, and will be used to determine:
  - The coordination latency of the coordination engine.
- "Single empty (null operation) task", and will be used to determine:
  - The average throughput and average latency involved in initiating a task.

- "Single long idle task", and will be used to determine:
    - The scalability of the engine with respect to coordinating large numbers of tasks.

1.1.2.2 Advanced transaction support

The performance and scalability of the advanced transaction support will be determined by performing a series of experiments. A set of "dummy transaction participants" will be implemented with differing temporal characteristics for their operations. For example, a participant could be implemented whose prepare operation takes around 5 seconds to complete. These "transaction participants" will be used in the following experiments:

- The performance of the advanced transaction support will be determined by measuring the system's average throughput and average latency. This experiment will be performed with different values of the following parameters:
    - Number of registered participants
    - Number of simulated clients (scalability)
    - Different "speed" participants
    - Different interaction scenarios, e.g. commit and rollback.
- The average latency associated with starting and terminating an "empty transaction" will be measured, and used to estimate the overhead involved in creating, starting and terminating.
- The average latency associated with registering participants with a transaction will be measured, and used to estimate the overhead involved in participant registration.

1.2 Strategy for Evaluating Functional Capabilities and Attributes

The strategy for evaluating the functional results of the ADAPT project has two complementary components. Firstly, to provide a proof-of-concept implementation of an application on top of the ADAPT platform, and secondly to identify the key technical goals of the project. Based on these goals, it is possible to compare the technical capabilities and attributes of the ADAPT approach with those of a selected set of existing and proposed technologies. These high-level comparisons lead to questions which can by used to evaluate the results of the ADAPT project. The key technical goals of the ADAPT project, in the area of functionality, have been identified as to make advances in support for:

- Service description
- Composability
- Configuration
- Adaptation
- Process definition
- Replication
- Security
- Transaction models

In the following sections each of ADAPT's key technical goals will be examined, and the functional capabilities and attributes of a selected set of technologies analysed. This enables the production of high-level questions which can be used to evaluate the functionality that the ADAPT project has derived.

1.2.1 Proof-of-concept implementation

The ADAPT functional evaluation will include a proof-of-concept implementation on top of the ADAPT platform, the purpose of this implementation is to validate the functionality provided by the ADAPT platform. The proof-of-concept implementation will be based on composing two different designs of a Supply Chain Management system: WS-I's (Web Services Interoperability) Sample Application Specification [34][35] and the RosettaNet's Order Management (Cluster 3) Standards [39][40]. For more details of the implementation see the ADAPT "Demonstrator Specification" [33].

1.2.2 Service description goal

Service description is an integral part of the overall service composition goals of the ADAPT project. Through service description the syntax and semantics of the required interactions with services can be published. The functional evaluation of the service description capabilities of the ADAPT platform is based around the analysis of the Java 2 Platform - Enterprise Edition (section A.1), CORBA Component Model (section A.2), Web Services Description Language (section A.13), OMG Interface Definition Language (section A.17), Web Services Choreography Interface (section A.9), Web Services Conversation Language (section A.12), CORBA Interface Repository (section A.18) and Web Services Reliable Messaging Protocol (section A.15). These technologies have a wide range of focuses including: interface specification, guaranteed delivery semantics and choreography specification. Analysing these technologies has contributed to the following evaluation questions:

- What is the additional information about ADAPT services that is required to support composition?
- Analysis of other technologies suggests that information about: interface, delivery semantics, transaction semantics, and choreography, should be provided. Is this addressed by ADAPT?

1.2.3 Composability goal

Composability is a goal for the ADAPT project at all level within the platform, not just for services. Without achieving some level on composability the complexity involved composition applications would become unmanageable. The functional evaluation of the composability capabilities of the ADAPT platform is based around the analysis of the CORBA Component Model (section A.2), GGF Grid, (section A.4) and Microsoft's .NET (section A.3). Analysing these technologies has contributed to the following evaluation questions:

- What is the additional support that ADAPT provides to enable composition of services?
- Analysis of other technologies suggests that support for: service discovery and long-lived transaction, are required. How are they supported by ADAPT?

1.2.4 Configuration goal

The configuration goal is based around providing the ability to specify the behaviour of ADAPT platform, in such a way that does not require code changes. This goal will facilitate composability and adaptability. The functional evaluation of the configuration capabilities of the ADAPT platform is based around the analysis of the Java 2 Platform - Enterprise Edition (section A.1), Microsoft's .NET, (section A.3) and Universal

Description, Discovery and Integration (section A.16). Analysing these technologies has contributed to the following evaluation questions:

- What is the additional support that ADAPT provides to enable configurable services?
- Analysis of other technologies suggests that support for system meta-information and meta-types are required. Are they supported by ADAPT?

1.2.5 Adaptation goal

The adaptation goal is based around providing the ability to dynamically specify the behaviour of ADAPT supported applications, in such a way that these applications can change to meet their new requirements. The functional evaluation of the adaptation capabilities of the ADAPT platform is based around the analysis of Java 2 Platform - Enterprise Edition (section A.1). Analysing these technologies has contributed to the following evaluation questions:

- What is the additional support that ADAPT provides to enable adaptation of services?
- Analysis of other technologies suggests that support for dynamic application update is required. Are these addressed by ADAPT?

1.2.6 Process definition goal

Process definition is an important part of the service composition goal of the ADAPT project. Through process definition the internal behaviour of composite services can be described. The functional evaluation of the process definition capabilities of the ADAPT platform is based around the analysis of Web Services Flow Language (section A.10), Microsoft's XLANG (section A.11) and Business Process Execution Language for Web Services (section A.8). Analysing these technologies has contributed to the following evaluation questions:

- What are the additional techniques that ADAPT provides to enable process definition?
- Analysis of other technologies suggests that support for: multiple conversations, compensation, exception handling, flow model and global model, are required. Are these addressed by ADAPT?

1.2.7 Replication goal

Replication is an important aspect of supporting the levels of service availability that ADAPT applications require. The functional evaluation of the replication capabilities of the ADAPT platform is based around the analysis of Microsoft Cluster Server (section A.20), IBM's WebSphere Clustering (section A.21), BEA's WebLogic Clustering (section A.22) and JBoss Clustering (section A.19). Analysing these technologies has contributed to the following evaluation questions:

- What is the additional support that ADAPT provides to enable replication?
- Analysis of other technologies suggests that support for: consistent replication in the context of transaction processing, state replication, cluster partitioning, dynamic cluster membership, load balancing and model cloning, are required. Are these addressed by ADAPT?

1.2.8 Security goal

Security is an important part of the service composition goal of the ADAPT project. Without security support composite services would not be able to span organisations. The functional evaluation of the security capabilities of the ADAPT platform is based around the analysis of Java 2 Platform - Enterprise Edition (section A.1) and Web Service Security (section A.14). Analysing these technologies has contributed to the following evaluation questions:

- What is the additional support that ADAPT provides to enable security?
- Analysis of other technologies suggests that support for: container managed security and message integrity, message confidentiality and access control, is required. Are these addressed by ADAPT?

1.2.9 Transaction models goal

Transaction models is an important part of the service composition goal of the ADAPT project. Without consistent transaction models the behaviour of composite services can't be ensured, in the presence of partial failure. The functional evaluation of the transaction modelling capabilities of the ADAPT platform is based around the analysis of Java 2 Platform - Enterprise Edition (section A.1), Business Transaction Protocol (section A.5), Web Services Coordination and Transaction (section A.6) and Web Services Composite Application Framework (section A.7). Analysing these technologies has contributed to the following evaluation questions:

- What is the additional support that ADAPT provides to enable transactional services?
- Analysis of other technologies suggests that support for: container managed transactions, non-atomic transactions, transaction qualifiers and long-lived activities, is required. Are these addressed by ADAPT?

2 References

[1] Sandeep Chatterjee and James Webber, “Developing Enterprise Web Services”, Prentice Hall, 2003.

[2] Java 2 Platform, Enterprise Edition Specification (http://java.sun.com/j2ee)

[3] OASIS Business Transaction Protocol (BTP), Committee Specification 1.0 (https://www.oasis-open.org/committees/business-transactions/)

[4] Web Services Transactions (WS-Transaction) (http://www.ibm.com/developerworks/library/ws-transpec/)

[5] Web Services Coordination (WS-Coordination) (http://www.ibm.com/developerworks/library/ws-coor/).

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

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

[8] Web Service Flow Language (WSFL) (http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf)

[9] Microsoft’s XLANG (http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm)

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

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

[12] Web Services Security 1.0 (WS-Security) (http://www-106.ibm.com/developerworks/webservices/library/ws-secure/)

[13] Universal Description, Discovery and Integration (UDDI) 3.0 Specification (http://www.uddi.org/specification.html)

[14] OMG Interface Definition Language (OMG IDL) (http://www.omg.org/cgi-bin/doc?formal/02-11-03)

[15] CORBA Interface Repository (CORBA IR) (http://www.omg.org/cgi-bin/doc?formal/02-11-03)

[16] Microsoft .Net (http://www.microsoft.com/net/)

[17] T. Thai and H.Q. Lam, “.NET Framework Essentials, Second Edition”, O’Reilly and Associates, ISBM 0-596-00302-1.

[18] CORBA Components (http://www.omg.org/cgi-bin/doc?formal/02-06-65)

[19] The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration (http://www.globus.org/research/papers/ogsa.pdf)

[20] The Open Grid Services Architecture (OGSA) Platform (http://www.ggf.org/ogsa-wg)

[21] Open Grid Services Infrastructure (OGSI) (http://www.ggf.org/documents/GFD/GFD-R-P.15.pdf)

[22] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) (http://msdn.microsoft.com/ws/2003/03/ws-reliablemessaging/)

[23] Web Services Composite Application Framework (WS-CAF) Primer (http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CAF-Primer.pdf)

[24] Web Service Context (WS-CTX) (http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf)

[25] Web Service Coordination Framework (WS-CF) (http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CF.pdf)

[26] Web Service Transaction Management (WS-TXM) (http://www.arjuna.com/library/specs/ws_caf_1-0/WS-TXM.pdf)

[27] Technical Overview of Clustering in Windows Server 2003 (http://www.microsoft.com/windowsserver2003/docs/ClusteringOverview.doc)

[28] Sacha Labourey and Bill Burke, “JBoss Clustering”, JBoss Group, August 2003.

[29] Özalp Babaoğlu, Alberto Bartoli, Vance Maverick, Alberto Montresor, Davide Rossi and Jakša Vučković, “JBoss Clustering Analysis”, ADAPT report, V1.1, March 2003.

[30] WebSphere Software Platform (http://www-3.ibm.com/software/info1/websphere/)

[31] BEA WebLogic Server (http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/se rver)

[32] Additional Structuring Mechanisms for the OTS Specification (http://www.omg.org/cgi-bin/doc?formal/2002-09-03).

[33] “Demonstrator Specification”, ADAPT project (IST-2001-37126), Deliverable D17, March 2004.

[34] “Supply Chain Management Sample Application Architecture”, WS-I, Version 1.0, December 2003 (http://www.ws-i.org/SampleApplications/SupplyChainManagement/2003-12/SCMArchitecture1.01.pdf).

[35] “Supply Chain Management Use Case Model”, WS-I, Version 1.0, December 2003 (http://www.ws-i.org/SampleApplications/SupplyChainManagement/2003-12/SCMUseCases1.0.pdf).

[36] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme and Gustavo Alonso, “Consistent Database Replication at the Middleware Level”, ACM Transactions on Computers, submitted.

[37] ECPerf Benchmark Specification (http://java.sun.com/j2ee/ecperf/)

[38] TPC Benchmark W (http://www.tpc.org/tpcw/default.asp)

[39] “RosettaNet Standards”, RosettaNet(http://www.rosettanet.org/standards).

[40] “RosettaNet Cluster 3: Order Management”, RosettaNet (http://www.rosettanet.org/Cluster3).

Appendix A Summary of Relevant Existing and Proposed Technologies

A.1 Java 2 Platform, Enterprise Edition (J2EE)

The Java 2 Platform, Enterprise Edition (J2EE) [2] is a collection a Java APIs that support commonly used distributed computing technologies and network services. The purpose of these APIs is to support the rapid development of Enterprise Applications. In general, the implementer of an Enterprise Application has many different issues to address, for example, providing Internet interfaces along with providing Intranet interfaces, also having to accessing legacy applications and databases.

A typical J2EE Enterprise Application provides its Internet interfaces via web pages, sometimes e-mail and in the future via Web services [1]. As for Intranet interfaces, J2EE supports both remote procedure call and message oriented integration with legacy applications and supports interaction with SQL based relational databases.

The J2EE APIs are numerous and provide an extensive set of functionality required to construct Enterprise Applications. In the rest of this section the following J2EE APIs will be briefly described:

- JDBC
- EJB
- Java IDL (CORBA)
- RMI
- JNDI
- JMS
- JTA
- Servlet/JSP
- JavaMail

The JDBC API allows a Java application to access relational databases, which may or may not be remotely hosted. Through the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to a database. The designers of JDBC have attempted to create a platform-neutral interface between database and application, where database vendors or third-party developers providing a JDBC driver, which is a set of classes that implements the required functions for a particular database.

Enterprise Java Beans (EJB) is a component model for units of business logic and business data. The EJB model attempts to separate the beans, their container and the server. The beans are provided by the application implementer, and three types of bean are supported: session, entity and message-driven. Session beans are used to encapsulate business logic; Entity beans are used to encapsulate business data; Message-driven beans are used to support asynchronous interaction with business logic. The EJB container manages a set of beans, handling such thing as: lifecycle management, instance pooling, distributed transaction management and security. The EJB server itself hosts the EJB containers.

The Java IDL API provides an interface between Java application and distributed object and services build using the Common Object Request Broker Architecture (CORBA). It is possible that such distributed objects and services could be legacy applications.

The Java Remote Method Invocation (RMI) API is Java's native scheme for creating and using remote objects. Java RMI is "Java native" because it deals directly with Java objects, this makes Java RMI particularly suitable for accessing remote EJBs.

The Java Naming and Directory Interface (JNDI) is an API that supports accessing naming and directory services from Java applications. JNDI is an integral part of the J2EE framework; it is used by J2EE components to access various runtime resources such as the transaction manager, EJB home references and JDBC data sources.

The Java Messaging Service (JMS) provides an API for Java applications to perform reliable asynchronous messaging. The JMS API is a portable interface between Java applications and a native message-oriented middleware (MOM) system. This allows Java applications to interact with legacy applications through the MOM system.

The Java Transaction API (JTA) allows Java applications to manage ACID transactions based on the X/Open XA API for distributed transactions.

The Java Servlet API and JavaServer Pages (JSP) provide a standard way to extend Web servers to support dynamic content generation. These two techniques though related provide different approaches dynamic content generation. Java Servlet API allows operations to be written which directly handle HTTP requests, and directly generate content, usually HTML, as a response to the request. It is not unusual for this generated content to contain information that has been obtained via EJBs from a database. JSPs provide a higher-level approach; a JSP is usually HTML or XML that has embedded within it special tags that cause fragments of Java to be executed. These fragments of Java can cause HTML or XML to be generated that is specific to the request.

The JavaMail API provides a platform and protocol-independent framework to build Java applications that utilise Internet e-mail. This allows J2EE Enterprise Applications to interact with users or other programs via e-mail.

As well as defining APIs, the J2EE standard also addresses issues such as deployment. The structure of entities called WAR (Web Application aRchive) and EAR (Enterprise Application aRchive) files are specified. WAR files contain static content such as HTML files, along with JSP and the Java class required to generate dynamic content. WAR files also contain a deployment descriptor, in XML, which specifies things such as how to map a URL to a servlet and how to map a servlet to a Java class. EAR files contain a collection of WAR and JAR files that make up an enterprise application, along with a deployment descriptor, in XML, which specifies such things as the "context root" associated with a WAR file. Most J2EE platforms support redeployment of EAR and WAR files, that is if an EAR or WAR needs to be updated, then all that is required is to replace the existing file. The change will be spotted by the J2EE platform and the old application will be replaced by the new version.

A.2 CORBA Component Model (CCM)

The CORBA Component Model (CCM) [18] is an OMG standard designed to extend the CORBA 2 model to address configuration and deployment. The CORBA Component Model first appeared in CORBA 3.0, and introducing the concept of components to CORBA. Components go beyond the existing concept of CORBA interfaces, which were specified CORBA 2's OMG IDL.

In the CORBA Component Model a component is modeled as a set of attributes and a set of ports. The attributes of a component are mainly intended to be used as configuration properties. These component attributes differ from OMG IDL attributes, in that, in the context of CCM, attributes may raise exceptions when misused. The ports of a component are of four types: facets, receptacles, event sinks and event sources.

- *Facets* are distinct named interfaces provided by the component for client interaction.
- *Receptacles* are named connection points that describe the component's ability to use a reference supplied by some external agent.
- *Event sources* are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel.
- *Event sinks* are named connection points into which events of a specified type may be pushed.

The use of facets allows the CCM to make a distinction between interface features that are used primarily for configuration, and interface features that are used primarily by application clients during normal application operation. A client can obtain the object references that correspond to the component's facets using the *equivalence interface*, which all components possess.

In the CCM a component possesses a meta-type, *home*, through which it can be managed. The home meta-type maps to a set of interfaces which allow component instances to be created and found.

Below is an example of a component description in CORBA 3's Interface Definition Language (IDL), the language used to describe components.

```
component BankAccount
{
    attribute Boolean         active;
    attribute BankAccountType type;

    provides CustomerFacet      customer;
    provides BankAssistantFacet bankAssistant;
    provides BankManagerFacet   bankManager;

    uses BankPlug bank;

    publishes OverdrawnEvt          overdrawn;
    emits     DirectDebitPaymentEvt directDebitPayment;
    consumes  InterestRateChangeEnv interestRateChange;
};
```

This example contains the specification of a component *BankAccount*; this component possesses:

- Two attributes *active* and *type*, of types *Boolean* and *BankAccountType* respectively.
- Three facets *customer*, *bankAssistant*, and *bankManager*, the interface exposed by these facets are of types *CustomerFacet*, *BankAssistantFacet*, and *BankManagerFacet* respectively.

- One receptacle *bank*, the interface exposed by this receptacle is of types *BankPlug*.
- Two event sources *overdrawn* and *directDebitPayment*, along with one event sink *interestRateChange*. Note that the event source *overdrawn* is indicated to be a publisher, which means that its events can go to multiple destinations, whereas the event source *directDebitPayment* is indicated to be an emitter, so only goes to one destination.

Below is an example of a home meta-type description in CORBA 3's IDL. This example contains a home meta-type called *BankAccountHome* for a component *BankAccount*.

```
home BankAccountHome manages BankAccount primaryKey BankAccountKey
{
    factory create_bank_account(in long id)
        raises (Components::DuplicateKeyValue, Components::InvalidKey);

    finder  search_bank_account(in long id)
        raises (Components::UnknownKeyValue, Components::InvalidKey);
};
```

The meta-type called *BankAccountHome* specified a primary key of *BankAccountKey* and describes a factory operation *create_bank_account* and a finder operation *search_bank_account*. Both the component and home meta-types are mapped to a set of IDL file, which contains the interfaces that the user will have to provide corresponding implementation. The relationship between the component's meta-types and their associated implementation is specified in OMG Component Implementation Definition Language (CIDL), an example of which is below:

```
composite entity BankAccountImpl
{
    home executor BankAccountHomeImpl
    {
        implements BankAccountHome;
        managers   BankAccountImpl;
    };
};
```

The CCM supports a set of mechanisms to support configuration. These mechanisms can be deployed in a number of ways in a component implementation or application. The configuration is based around setting the attributes associated with the component. These attributes can be set using *configuration objects* that encapsulate the configuration attributes, and is used configure any instance of the component it is applied to. Another mechanism is to use *factory based configuration*, by which when an instance of the component is created by the factory, the factory sets the attributes. The home meta-type of a component may support the *HomeConfiguration* interface, which allow interaction with it's configuration information.

In order to allow deployment of CORBA component, its implementations, a software package descriptor and other files, needs to be assembled into a *software package* (component package), in the form of a ZIP archive file. The contents of a software package are described by a software package descriptor. The descriptor consists information about the software followed by one or more sections describing implementations of that software. Such software package descriptors are written in CORBA Software Descriptor (CSD), which is encoded in XML.

A *component assembly package* is the mechanism for deploying a set of interrelated component implementations. A component assembly package contains a *component assembly descriptor*, which describes what components make up the assembly, how those components are partitioned, and how they are connected to each other. A component assembly descriptor is the recipe for deploying a set of interconnected components.

A.3 Microsoft's .NET

The Microsoft .NET initiative was announced in July 2000 [16][17]. The Microsoft .NET platform is an application development framework that provides tools, services and APIs to support the construction of sophisticated applications.

One of the key goals to .NET initiative was to facilitate better interoperability between libraries and components written in different programming languages. To allow this the .NET platform supports the Common Language Runtime (CLR), which manages and executes code written in .NET languages. Microsoft and its partners have provided CLR compatible versions of many popular languages, for example, Managed C++, Visual Basic.NET (VB.NET) and C#. C#, which is similar to Java, has been designed to take advantages of CLR features, such as garbage collection, exception management, and reflection. .NET language compilers produce Microsoft's Intermediate Language (IL), which is loaded and verified by the CLR, and when required to be executed it is passed to a "just in time" compiler to be converted to executable machine instructions. A key element of .NET languages is their support for attributes. Attributes are names, and optionally values, which can be placed in the code, and associated with classes or operations. Attributes allow the infrastructure to automatically inject code at runtime, to perform the required effects.

A part of the .NET platform is a set of *.NET Enterprise Server* products that shorten the time required to develop large-scale business systems. These products include Application Center 2000, BizTalk Server 2000, Commerce Server 2000, Exchange Server 2000, Host Integration Server 2000, Internet Security and Acceleration 2000, and SQL Server 2000. These services are highly reusable, and can be easily integrated into new applications. Along with these services the .NET platform also provides sets of classes to support particular types of application needs. The ADO.NET classes enable developers to interact with data from databases, as XML. The ASP.NET classes support the development of Web-based applications and Web services. The Windows Forms classes support the development of desktop-based client applications.

Support for Web services is an important aspect of .NET. Programming Web services has been make comparatively easy, all that is required to allow functionality to be exposed as a Web service is to add an attribute to the code that provides the implementation to indicate that it is a Web service, and add attributes to the methods which are to be exposed by the Web service. Web service discovery is supported by the use of DISCO files. DISCO files contain elements that refer to a set of Web service's endpoint and WSDL, a DISCO file can also contain an element that exposes all Web services below a particular subdirectory.

A.4 GGF Grid

Grid technologies are based around providing support for the sharing and coordinated use of geographically and organizationally distributed resources. To support standardization in the area Grid technologies the Global Grid Forum (GGF) has set up a number of working groups whose purpose is to provide specifications, guidelines and recommendations. The results of one such working group, the Open Grid Service

Architecture Working Group (OGSA-WG), is the Open Grid Service Architecture (OGSA) Platform [19][20]. This is an attempt to standardise the approaches to and mechanisms for solving basic problems common to many Grid systems, namely: communicating with other services, establishing identity, negotiating authorisation, service discovery, error notification and managing service collections

The OGSA Platform has three principal elements: the Open Grid Services Infrastructure (OGSI), the OGSA Platform Interfaces, and the OGSA Platform Models.

- Open Grid Services Infrastructure (OGSI) [21] defines mechanisms for creating, managing and exchanging information among Grid services. Grid services are Web services that conform to a set of conventions that define how a client interacts with a Grid service. These conventions cover both how the Grid service behaves and the interfaces the Grid service provides. Grid service interfaces are described in WSDL, with OGSI extensions (GWSDL) to allow: extension of Web services interfaces, asynchronous notification of state change, references to instances of services, collections of service instances, and service state data that augments the constraint capabilities of XML schema definition.
- *OGSA Platform Interfaces* build on OGSI mechanisms to define interfaces and associated behaviors for various functions not supported directly within OGSI, such as service discovery, data access, data integration, messaging, and monitoring.
- *OGSA Platform Models* support these interface specifications by defining *models* for common resource and service types.

One of the notable difference between non-Grid Web services and Grid services is that Grid services use Grid Service Handles (GSH) and Grid Service References (GSR) to gain access to a Grid service instance, whereas, non-Grid Web services tend to use URIs to address services. A GSH can be regarded as a network-wide pointer to a specific Grid service instance. The GSH does not carry enough information to allow a client to communicate directly with the service instance. Instead, a client wishing to communicate with a Grid service instance must resolve the GSH to a GSR. The Grid service instance is then accessible to a client application through the use of the GSR.

Unlike non-Grid Web services, Grid service instances have a well-defined life-cycle. Grid service instances can be created then later destroyed. A client may request the creation of a Grid service instance by invoking the *createService* operation on a Grid service instance that implements a port type that extends the *Factory* port type, which is defined by the OGSI.

OGSI provides a mechanism, Service Data, to allow the exposing of a Grid service instance's state data to service requestors for query, update and change notification. The publicly accessible state of a Grid service is declared as part of the service's GWSDL definition. This is roughly equivalent to the idea of declaring the attributes that are possessed by an object. OGSI provides extensible operations for querying, updating, and subscribing to notification of changes, of a Grid service's Service Data as that data evolves over the lifetime of the Grid service.

OGSI requires all Grid service instances to implement the GridService port type. This port type is analogous to the base "Object class" in many object-oriented programming languages, in that it encapsulates the base behavior of a Grid service instance. The GridService port type allow the querying and updating of Service Data set of the Grid service instance and managing the termination of the instance.

A.5 Business Transaction Protocol (BTP)

The Business Transaction Protocol (BTP) [3] is an OASIS specification designed to provide reliable coordination of parties engaged in a business-level transaction.
Ratified to committee specification level in May 2002, BTP has the backing of several large IT organizations including HP, BEA, Sun, and Oracle as well as a number of small and medium-sized vendors.

BTP provides a common understanding and a way to communicate levels of participation within transactions and limits on these levels between organizations. The formal rules are necessary for the distribution of parts of business processes outside the boundaries of an organization. BTP solves part of the problem for developers of loosely coupled transactions—the coordination of services/participants to ensure a consistent termination outcome. Expertise in the design of compensating actions is still required, but these compensations are local rather than distributed.

BTP uses a two-phase completion protocol for transaction coordination. At termination time, services participating in a transaction are asked up-front to state their intention (whether they will proceed with the transaction or whether they are not prepared to do so), and will later be instructed by the transaction manager to either proceed or not based on the analysis of all of the collected intentions. In order to satisfy its requirements, BTP supports two distinct transactions models, which are:

- Atoms: Similar to traditional atomic transactions where all Web services participating in an Atom are *guaranteed* to see the same outcome as all of the other participants: the outcome is atomic.
- Cohesions: Which allow business logic to dictate which combination of participating services succeed, while permitting the transaction as a whole to make forward progress – this allows the business transaction to proceed even in the presence of failures.

Cohesions allow us to pick groups of participating services (known as the *confirm set*) that we would eventually like to reach completion: unlike an atom, not all participants need see the same outcome for the transaction, i.e., atomicity is relaxed.

The BTP transaction coordinator is not as dictatorial as its equivalents in other transaction management models: the BTP model recognizes the possibility that other parts of an application might need to influence the decision making process required to complete a transaction. In addition, BTP supports runtime negotiation of quality of service characteristics based on the exchange of *qualifiers*. Qualifiers are the mechanism which enables the bilateral exchange of protocol “small print” between participants and coordinators. In essence, each BTP message allows the sender to tag qualifiers that describe such things as, “I will be prepared for the next ten minutes, and after that I will unilaterally cancel” and “You must be available for at least the next 24 hours to participate in this transaction.” Qualifiers are a valuable mechanism in Web services transactions because in a loosely coupled environment, it is extremely useful to know that the party you’re communicating with will only be around for so long, or to be able to specify that your party won’t hang around while others procrastinate.

A.6 Web Services Transaction (WS-Transaction) and Web Services Coordination (WS-Coordination)

The purpose of the combined Web Services Transaction (WS-Transaction) [4] and Web Services Coordination (WS-Coordination) [5] specifications is to provide a standard for conducting transactions over Web services. The WS-Transaction and WS-Coordination specifications were published by IBM, Microsoft and BEA in August 2002, and are intended to be a competitor to the Business Transaction Protocol (BTP) [3] specification.

The current WS-Transaction and WS-Coordination specifications are lacking details that would be required to construct interoperable implementation, which are based purely on these specifications. The authors of these specifications have stated that it is their intention to submit them to a standards body such as W3C or OASIS.

The WS-Coordination specification describes an extensible framework for providing protocols that coordinate the actions of distributed applications. The framework consists of two simple message-oriented request-response protocols for *Activation* and *Registration*. Being message-oriented the request and response are separated into two one-way invocations.

The *Activation protocol* specifies how an application can request a coordination context from an activation service, for a specified coordination type. The returned coordination context will contain, at least, a unique identifier for the context, the coordination type, and the endpoint address of the registration service for that context. This coordination context can be passed between applications so allowing other application to register with the registration service for the context.

The *Registration protocol* specifies how an application can request that a registration service includes the application as a participant in the protocol associated with the context. As part of the registration protocol the application sends to the registration service its participant endpoint address, in response the registration service responds with the context's protocol coordinator's endpoint address.

Building on WS-Coordination, the WS-Transaction specification describes two coordination types Atomic Transactions and Business Activities. Associated with the Atomic Transaction coordination type are five simple message-oriented protocols for *Completion*, *Completion with Acknowledgement*, *Phase Zero*, *Two-Phase Commit* and *Outcome Notification*. Associated with the Business Activity coordination type are two simple message-oriented protocols for *Business Agreement* and *Business Agreement with Complete*.

The Atomic Transaction protocols provide support for standard all-or-nothing ACID transactions. Through the *Completion* and *Completion with Acknowledgement* protocols participants can request the terminations, commit or abort, of a transaction. If "with acknowledgement" the participant must acknowledge receipt of the final outcome before the corresponding coordinator can safely forget the transaction. The final outcome of a transaction can also be requested using the *Outcome Notification* protocol. If a participant wishes to participate in the transaction as a resource the *Phase Zero* and *Two-Phase Commit* protocols are available.

The Business Activity protocols provide support for handling long-lived activities and the desire to apply business logic to handle business exceptions. The protocols allow a coordinator to control a participant within a business activity, for example, the coordinator can request that the activity abandon its work in some appropriate way, or after completion the coordinator can request compensation be performed.

A.7 Web Services Composite Application Framework (WS-CAF)

Web Services Composite Application Framework (WS-CAF) [23] is a set of three related specifications aimed at solving problems that arise when combining multiple Web services, also known as "composite applications." Composite applications require specific support services, such as the ability to share common information, or context, and the ability for the success or failure of individual Web services to be tied to the success or failure of the larger unit of work comprising the set Web services. The Web Services Composite Application Framework specifications were published by Arjuna Technologies, Fujitsu, IONA Technologies, Oracle, and Sun Microsystems in July 2003.

WS-CAF is aimed at providing the common information or shared context of a composite Web services application. This can include such items as security credentials, so that someone can log in once and invoke multiple Web services without having to log in again; a database connection over which to perform multiple operations from multiple Web services without having to establish a new connection each time, or a device address to which to post results from multiple Web services without having to search for the device's network address each time.

Another aim of WS-CAF is to also allow the mapping of the success and failure of individual Web services into success or failure of the web service application as a whole. This is achieved by means of the ability to group multiple Web services into a single transaction with various, configurable properties to ensure that composite operations reliably produce a known state regardless of the failure of one or more Web services to successfully complete. Furthermore, the specifications support multiple transaction protocols, including long running actions with compensations and asynchronous business process flows.

The three specifications that comprise the WS-CAF are:

- Web Services Context (WS-CTX) [24], which models a Web services context data structure as a Web resource, accessible via standard URLs.
- Web Services Coordination Framework (WS-CF) [25], which defines a software agent called a coordinator that takes responsibility for context management and augmentation. Web services in a composite application register with a coordinator to ensure message results are communicated.
- Web Services Transaction Management (WS-TXM) [26], which defines three distinct transaction protocols that can be plugged into the coordination framework for interoperability across existing transaction managers, long running compensations, and business process automations.

The WS-CAF specification states that the overall aim of the combination of the parts of WS-CAF is to support various transaction processing models and architectures. It also states that the individual parts of WS-CAF are designed to complement Web services orchestration and choreography technologies such as BPEL4WS, WSCI and WS-Choreography and work with existing Web services specifications such as WS-Security and WS-Reliability. The three specifications define incremental layers of functionality that can be implemented and used separately by these and other specifications separately or together. The emphasis of WS-CAF is to define supporting services required by Web services used in combination.

A.7.1 WS-Context (WX-CTX)

WS-Context provides the ability to scope arbitrary units of distributed work by sharing a common context. This is a requirement in a variety of distributed applications, such as choreography and business-to-business interactions. Scoping makes it possible for Web services participants to be able to determine unambiguously whether or not they are in the same composite application, and what it means to share context. Scopes can be nested to arbitrary levels to better delineate application work.

The shared context allows a series of operations to share a common outcome. The Web services participating in the composite application maintain the context information shared between multiple participants in a Web services interaction.

The context is modelled as a Web resource and is accessible via a URI. Web services are identified as participants in the activity by including the context URI in the SOAP header, as shown in the following example:

```
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
    <env:Header>
    <n:Composite xmlns:n="http://example.org/CompositeApplication">
        <n:Context>http://example.org/contextURI</n:Context>
    </n:Composite>
    </env:Header>
    <env:Body>
        <m:Message xmlns:m="http://example.org/MessageSchema">
            ...
        </m:Message>
    </env:Body>
</env:Envelope>
```

Web services can also choose to join a composite application upon receipt of a SOAP message containing the context URI in the header, or, optionally, containing the context itself within the body of the SOAP message. In this way the context service supports passing the context by reference (a URI in the header) or by value (within the SOAP body).

A Context Service is modelled as a Web service that can be co-located with the participants or executed as a separate service. All Web services referencing the same context URI, or accepting the same context in a SOAP message, are considered part of the same *composite application*. An implementation may permit the Web services to manage the context themselves, or use a separate service to manage the context.

The context contains information, including a unique context identifier (URI), the URI of the initial requester, the URI of the ultimate provider, optional URIs of each additional Web service within the scope of the composite application, optional security information, and a status result for each Web service in the composite application.

A.7.2 WS-CoordinationFramework (WS-CF)

In WS-CoordinationFramework coordination is the act of one agent (the *coordinator*) disseminating information to a number of *participants* to guarantee that all participants obtain a specific message. A coordinator can accept the responsibility, for example, of notifying all participants in an activity of a common outcome. Coordination is a fundamental requirement in distributed systems that many applications use either explicitly or implicitly, e.g., workflow, atomic transactions, caching and replication, security, auctioning, and business-to-business activities. Coordination propagates additional information (the *coordination context*) to the participants.

The WS-CF allows the *management and coordination* in a Web services interaction of a number of *activities* related to an overall application. It builds on the Web Services Context Service specification and provides a coordination service that plugs into WS-Context. In particular WS-CF:

- Defines demarcation points which specify the start and end points of coordinated activities; this is done automatically by invoking an Activity;
- Defines demarcation points where coordination of participants occurs (i.e., at which points the appropriate SOAP messages are sent to participants);
- Registers participants for the activities that are associated with the application;
- Propagates coordination-specific information across the network by enhancing the default context structure provided by WS-CTX;

The main components involved in using and defining the WS-CF are:

- A *Coordinator*: Provides an interface for the registration of participants (such as *activities*) triggered at *coordination points*. The coordinator is responsible for communicating the outcome of the activity to the list of registered activities. Importantly, coordination is not restricted to the end of an activity: an activity can execute (different) coordination protocols at arbitrary points during its lifetime. Coordination extends the notion of an *activity* to represent a defined set of tasks with a set of related *coordination actions*;
- A *Participant*: The operation or operations that are performed as part of coordination sequence processing
- A *Coordination Service*: Defines the behavior for a specific coordination model. The Coordination Service provides a processing pattern that is used for outcome processing. For example, an ACID transaction service is one implementation of a Coordination Service that provides a two-phase protocol definition whose coordination sequence processing includes *Prepare*, *Commit* and *Rollback*. Other examples of Coordination Service implementations include extended transaction patterns such as Sagas, Collaborations, Nested or Real-Time transactions and non-transactional patterns such as Cohesions and Correlations. Coordination can also be used to group related non-transactional activities. Multiple Coordination Service implementations may co-exist within the same application and processing domain. WS-CF does not specify how a Coordination

Service is implemented. For example, a given implementation may support multiple coordination protocols as in [30].

A.7.3 WS-Transaction Management (WS-TXM)

Generally structuring mechanisms available within traditional transaction systems are sequential and concurrent composition of transactions. These mechanisms are sufficient if an application function can be represented as a single top-level transaction. Frequently with Web services this is not the case. Top-level transactions are most suitably viewed as "short-lived" entities, performing stable state changes to the system; they are less well suited for structuring "long-lived" application functions, ones that could running for minutes, hours, days or even longer. Long-lived top-level transactions implemented using traditional systems may reduce the concurrency in the system to an unacceptable level by holding on to locks for a long time; further, if such a transaction rolls back, much valuable work already performed could be undone. Web services, because of their inherently unpredictable invocation patterns do not fit well with traditional ACID systems.

There are three transaction protocols defined by WS-TXM:

- *ACID transaction*: a traditional ACID transaction (AT) designed for interoperability across existing transaction infrastructures.
- *Long running action*: an activity, or group of activities, which does not necessarily possess the guaranteed ACID properties. A long running action (LRA) still has the "all or nothing" atomic effect, i.e., failure should not result in partial work. Participants within an LRA may use forward (compensation) or backward error recovery to ensure atomicity. Isolation is also considered a back-end implementation responsibility.
- *Business process transaction*: an activity, or group of activities, that is responsible for performing some application specific work. A business process (BP) may be structured as a collection of atomic transactions or long running actions depending upon the application requirements.

The long running action model (LRA) is designed specifically for those business interactions that occur over a long duration. Within this model, an activity reflects business interactions: all work performed within the scope of an application is required to be compensatable. Therefore, an application's work is either performed successfully or undone. How individual Web services perform their work and ensure it can be undone if compensation is required, are implementation choices and not exposed to the LRA model. The LRA model simply defines the triggers for compensation actions and the conditions under which those triggers are executed.

In the LRA model, each application is bound to the scope of a compensation interaction. For example, when a user reserves a seat on a flight, the airline reservation centre may take an optimistic approach and actually book the seat and debit the users account, relying on the fact that most of their customers who reserve seats later book them; the compensation action for this activity would obviously be to un-book the seat and credit the user's account. Work performed within the scope of a nested LRA must remain compensatable until an enclosing service informs the individual service(s) that it is no longer required.

In the business process transaction model (BP model) all parties involved in a business process reside within *business domains*, which may themselves use business processes to perform work. Business process transactions are responsible for managing interactions *between* these domains. A business process (business-to-business interaction) is split into *business tasks* and each task executes within a specific business domain. A business domain may itself be subdivided into other business domains (business processes) in a recursive manner.

Each domain may represent a different transaction model if such a federation of models is more appropriate to the activity. Each business task (which may be modelled as a scope) may provide implementation specific counter-effects in the event that the enclosing scope must cancel. In addition, periodically the controlling application may request that all of the business domains checkpoint their state such that they can either be consistently rolled back to that checkpoint by the application, or restarted from the checkpoint in the event of a failure.

An individual task may require multiple services to work. Each task is assumed to be a compensatable unit of work. However, as with the LRA model described earlier, how compensation is provided is an implementation choice for the task.

A.8 Business Process Execution Language for Web Services (BPEL4WS)

The purpose of Business Process Execution Language for Web Service (BPEL4WS) [6] is to provide a standard for specifying business process behaviour and business process interactions, for applications composed from Web services. The BPEL4WS 1.0 specification was published by IBM, Microsoft and BEA in July 2002, and is intended to supersede the XLANG [9] and WSFL [8] specification. The BPEL4WS 1.1 was published May 2003, and was used as input to an OASIS standardization working group.

The BPEL4WS integration model is that 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. The conversations are coordinated within a partner by a business process.

From the business process perspective, the services provided by other partners and the services expected by partners are specified as a set of partner links. The partner links, an extension of WSDL [11], are used to model the peer-to-peer partner relationships. Partner links define the shape of a relationship with a partner by defining the messages and interfaces (WSDL port types) used in the interactions in both directions.

Associated with each business process instance is a state, this state is comprised of a set of variables that contain messages. Variables can be used as the destination of received message or invocation results or the source of a reply message or invocation parameters. The contents of a variable can be also be accessed by certain basic activities, such as assign and switch.

The behaviour of a business process is specified using a set of activities. The execution of theses activities is structured using certain “structured activities”: sequence, switch, pick, while, flow and scope.

- sequence activity: this activity contains one or more activities that are executed sequentially;
- switch activity: this activity contains an ordered list of one or more conditions and activity pairs. The conditions are considered in order and the first condition that evaluates to true has its associated activity executed. In the case where no condition holds true a default “otherwise activity” can be specified;
- while activity: this activity repeatedly executes an activity while its condition is true.
- pick activity: this activity contains an ordered list of one or more event and activity pairs. It awaits the occurrence of these events, and then executes the associated activity. Only one activity will be executed, even if multiple applicable events occur. The events which can be monitored are either “message events” or “alarm events”.
- flow activity: this activity provides concurrent execution the set of contained activities. These activities, and their sub-activities, can be “linked” to form “must occur before” relationships between activities.
- scope activity: this activity allows the contained activity to be associated with its own fault handlers and compensation handler.

To support communication with external Web services, three communication activities are provided: receive, reply, invoke.

- receive activity: this activity is used to wait for a particular message type from a particular partner, and place the contents of the message into a container. A receive activity can be flagged to create a process instance.
- reply activity: the activity is used to send a response to a request previously accepted through a receive activity, the contents of the message being obtained from a container.
- invoke activity: this activity can be used to invoke a service (either synchronous or asynchronous) provided by a partner. The request message will be obtained from a container, and if the invoke activity is synchronous, the response will be stored in a container.

Other activities include assign, wait and empty, and error handling activities: throw, terminate and compensate.

- assign activity: the activity can be used to assign (parts of) a message contained within one container to an other container.
- wait activity: the activity is used to introduce delays for a certain period of time or until a certain deadline is reached.
- empty activity: the activity can be used to introduce an activity that “does nothing.” This activity can be useful within compensation handlers that are not required to perform any changes.
- throw activity: the activity is used when a business process needs to signal an internal fault explicitly.
- terminate activity: the activity can be used to immediately abandon all execution within the current business process instance.

- compensate activity: the activity can be used to cause the initiation of compensation on a scope that has already completed its execution normally.

BPEL4WS does not assume that each business process instance or service instance has a distinct “endpoint,” or that interactions are based on a sophisticated transport infrastructure that can identify the involved participants, instead it provides support for “message correlation.” Message correlation enables message contents to be examined to identify involved business process, even over multiple conversations. If interactions are based on a sophisticated message transport, message correlation may not be required.

A.9 Web Services Choreography Interface (WSCI)

The Web Service Choreography Interface (WSCI) [7] 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. The WSCI specification was authored by BEA Systems, Intalio, SAP AG and Sun Microsystems.

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 modelling 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.

- Interfaces: The behaviour of a Web service is described as processes that are contained within the interface. A Web service may expose multiple interfaces for supporting multiple scenarios.
- Activities and choreographs of activities: The behaviour of the processes of interfaces is described in terms of choreographed activities. Activities can be atomic or complex. Atomic activities represent basic units of behaviour, such as sending and receiving messages. Complex activities are recursively composed from other activities. Complex activities support specific kinds of activity choreograph, such as: sequential execution, parallel execution, looping and conditional execution.
- Processes and units of reuse: The named units of behaviour in WSCI are processes, and are described by activities. Process can be reused, by referencing their names.
- Properties: References to values within the interface definition are modelled by properties. They are the equivalent of variables in imperative programming languages.
- Context: Scopes containing activities are modelled by contexts. They manage such thing as exception handling, and property definitions.
- Message correlation: To model the interrelationship between conversations WSCI has introduced message correlation. Different conversations can be distinguished by correlation instances, which are a set of properties' values.

- Exceptions: To model exceptional behaviour WSCI supports the definition of sets of activities that will be executed in response to particular exceptional behaviour.
- Transactions and compensation activities: WSCI contexts can be associated with a transaction. WSCI supports two models of transactional behaviour, atomic and open-nested. Atomic transactions have the standard ACID transaction properties. Open-nested transactions are composed of other transactions, which can be themselves atomic or open-nested transactions. The rollback of open-nested transactions is achieved by executing compensation activities.
- Global model: WSCI also allows modelling a multi-participant view of the overall message exchange by means of a global model. A global model is described by a collection of interfaces of the participant services, and a collection of links between the operations of the participant services.

A.10 Web Services Flow Language (WSFL)

The Web Services Flow Language (WSFL) [8] is an XML language for the description of Web services compositions. The WSFL 1.0 specification was published by IBM in May 2001, and has how been superseded by the BPEL4WS specification [6].

WSFL is intended to support two approaches to modelling Web service composition: Flow Models and Global Models. A Flow Model of composition is based on describing how to use the functionality provided by a collection of Web services. WSFL models these compositions by specified the flow of control and data between Web services. A Global Model of composition is based on describing how a collection of Web services interacts. The interactions are modelled as links between endpoints of the Web services' interfaces, each link corresponding to the interaction of one Web service with an operation of another Web service's interface. Both Flow Models and Global Models support recursive composition, where Web service compositions can itself be provided as a Web service.

The WSFL specification describes the concepts of its models using a metamodel. The metamodel describes the entities that make up the models, and their interrelationships.

A.11 Microsoft's XLANG

The purpose of Microsoft's XLANG specification [9] is to model the message exchange behaviour among Web services, and is expected to serve as the basis for an automated protocol engines that can track the state of process instances and help enforce protocol correctness in message flows. The XLANG specification was published by Microsoft in 2001 and is widely deployed as part of the BizTalk orchestration server, but has how been superseded by the BPEL4WS specification [6].

The goal of XLANG is to make it possible to formally specify business processes as stateful long-running interactions. The specific of such business processes are done in terms of the following:

- Sequential and parallel control flow constructs.
- Long running transactions with compensation.
- Custom correlation of messages.

- Flexible handling of internal and external exceptions.
- Modular behaviour description.
- Dynamic service referral.
- Multi-role contracts.

The XLANG approach to service description is to extend WSDL [11] service descriptions with extension elements that describe the behavioural aspects of the service. An example of such a behavioural description is:

```
<wsdl:service name="TransferService">
    . . .
    <xlang:behaviour xmlns:xlang="http://schemas.microsoft.com/biztalk/xlang/">
        <xlang:body>
            <xlang:sequence>
                <xlang:action operation="Debit" port="BankAccountA" activation="true"/>
                <xlang:action operation="Credit" port="BankAccountB"/>
            </xlang:sequence>
        </xlang:body>
    </xlang:behavior>
</wsdl:service>
```

A.12 Web Services Conversation Language (WSCL)

The purpose of Web Service Conversation Language (WSCL) [10] 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 specification was published as a W3C note by Hewlett-Packard in March 2002.

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).

A.13 Web Services Description Language (WSDL)

The purpose of the Web Services Description Language (WSDL) [11] is to address the need to describe network services. In WSDL, network service descriptions are contained in XML documents, which defines both the abstract and concrete entities required to specify network services in sufficient detail that they can be invoked, and related to other network service.

The entities, which are encoded as XML elements within a WSDL document, are:

- *types*: contain data type definitions using some type system, normally XML Schema.
- *messages*: are abstract definition of the content of messages being communicated.
- *port types*: are abstract definition of a set of operations supported by one or more endpoints.
- *bindings*: contains the specification of concrete protocol and data format to communicate.

- *services*: contain the specification of a collection of endpoints.

An illustration of the overall structure of a WSDL document, is given below:

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

<wsdl:definitions name="..." targetNamespace="..." ...>
    <wsdl:types>...</wsdl:types> ?
    <wsdl:message name="...">...</wsdl:message> *
    <wsdl:portType name="...">...</wsdl:portType> *
    <wsdl:binding name="..." type="...">...</wsdl:binding> *
    <wsdl:service name="...">...</wsdl:service> *
</wsdl:definitions>
```

In the rest of this section the WSDL document elements, and their sub-elements, will be described in more detail.

The purpose of the *types* elements within a WSDL document is to allow the definitions of data types that can be included within *messages*. This is usually done using an XML Schema, though it should be noted that eventual wire format messages might not be in XML. The optional *types* element may be augmented by importing existing schemas into the WSDL document. An example of a WSDL document's types element is:

```
<wsdl:types>
    <xsd:scheme targetNamespace="http://example.com/job-control">
        <xsd:complexType name="NameValuePairType">
            <xsd:sequence>
                <xsd:element name="Name"  type="xsd:string"/>
                <xsd:element name="Value" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:scheme>
</wsdl:types>
```

Message elements are abstract definition of messages, which are exchanged between endpoints. Messages consist of zero or more logical *parts*, which are analogous to named and typed parameters. Each *part* is associated with a type from some type system, possibly defined in the *types* elements. An example of a WSDL document's message element is:

```
<wsdl:message>
    <wsdl:part name="AccountNumber" type="tns:AccountNumberType"/>
    <wsdl:part name="Amount"        type="xsd:integer"/>
</wsdl:message>
```

The purpose of the *port type* elements within a WSDL document is to allow the definition of abstract endpoints. The abstract endpoint definition is in terms of a set of *operation* elements. Each *operation* elements defines it name, and the message types involved in the operation. Depending on the messages involved in the operation four interaction models are supported:

- One-way: the endpoint receives a message.
- Request-response: the endpoint receives a message, and sends a correlated message.
- Solicit-response: the endpoint sends a message, and receives a correlated message.
- Notification: The endpoint sends a message.

Examples of port type elements, which are describing respectively: one-way, request-response, solicit-response and notification, are given below:

```
<wsdl:portType name="JobControl">
    <wsdl:operation name="Shutdown">
        <wsdl:input message="tns:ShutdownRequest"/>
    </wsdl:operation>

    <wsdl:operation name="GetStatus">
        <wsdl:input  message="tns:GetStatusRequest"/>
        <wsdl:output message="tns:GetStatusResponse"/>
        <wsdl:fault name="unknownService" message="tns:ErrorResponse"/>
    </wsdl:operation>

    <wsdl:operation name="AreYouAlive">
        <wsdl:output message="tns:AreYouAliveRequest"/>
        <wsdl:input message="tns:AreYouAliveResponse"/>
        <wsdl:fault name="unknownClient" message="tns:ErrorResponse"/>
    </wsdl:operation>

    <wsdl:operation name="Refresh">
        <wsdl:output message="tns:RefreshRequest"/>
    </wsdl:operation>
</wsdl:portType>
```

The purpose of the *bindings* elements are to define, for a particular *port type*, protocol details for operations, and the message format for associated messages. The *bindings* elements are extensible in that elements for other schemas can be introduced within the bindings. Specifications exist for bindings for SOAP, HTTP and MIME. Through these extensions the specifics of the protocol and message format can be specified for that particular from of binding. In the example below the SOAP binding extensions have been used:

```
<wsdl:binding name="JobControlSOAPBinding" type="tns:JobControl">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetStatus">
        <soap:operation soapAction=""/>
        <wsdl:input>
            <soap:body use="encoded" namespace="http://example.com/job-control"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </wsdl:input>
        <wsdl:output>
            <soap:body use="encoded" namespace="http://example.com/job-control"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </wsdl:output>
    </wsdl:operation>

    ...
</wsdl:binding>
```

In this example the protocol transport has been specified as HTTP, and that style of messages is RPC (which embeds messages within an element with the same name as the intended receiving method, as opposed to document which does not) and the messages will be encoded using SOAP encoding (which uses the SOAP schema as opposed to literal encoding where other arbitrary schemas are used).

The purpose of the *service* element is to define a set of network endpoints. Each network endpoint is defined by a *ports* element. In the example bellow, a service is specified containing a single network endpoint that is accessible at the specified URL, and is associated with a SOAP binding.

```
<wsdl:service name="JobControlService">
    <wsdl:port name="JobControl" binding="tns:JobControlSOAPBinding">
        <soap:address location="http://ws.acme.com/axis/services/JobControlService"/>
    </wsdl:port>
</wsdl:service>
```

A.14 Web Service Security (WS-Security)

The Web Service Security (WS-Security) [12] provides a framework to enable applications to construct secure SOAP message exchanges. The WS-Security 1.0 specification was published by IBM, Microsoft and VeriSign in April 2001.

The WS-Security specification describes enhancements to SOAP to allow message integrity and message confidentiality to be implemented. The WS-Security enhancements are aimed to addressing two types of security threat: 1) messages being modified or read by antagonists or 2) an antagonist sending messages to a service that, while well-formed, lack appropriate security claims to warrant processing.

The WS-Security approach is to define the basic structure of a SOAP header for attaching security information to a SOAP message, and specifying of how encrypted information is encoded in a SOAP message. The types of security information that can be attached to SOAP message in a WS-Security SOAP header are:

- User name and password information (UsernameToken element)
- Binary security tokens, for example, X.509 certificates and Kerberos tickets (BinarySecurityToken element)
- References to security tokens what reside somewhere else and need to be "pulled" by the receiving application (SecurityTokenReference element)
- Key information, such as X.509 certificates (KeyInfo element)
- Message signature (Signature element)

A.15 Web Services Reliable Messaging Protocol (WS-ReliableMessaging)

The Web Services Reliable Messaging Protocol (WS-ReliableMessaging) [22] describes a protocol to allow messages to be delivered reliably between two distributed applications in the presence of software component, system and network failures. The Web Services Reliable Messaging Protocol specification was published by BEA, IBM, Microsoft and TIBCO Software in March 2003.

The WS-ReliableMessaging protocol guarantees that messages sent by the initial sender will be delivered to the ultimate receiver. This guarantee is specified by delivery assurances, the four basic delivery assurances that an endpoint can provide are:

- *AtMostOnce*: Messages will be delivered at most once without duplication or an error will be raised on at least one endpoint. It is possible that some messages in a sequence may not be delivered.
- *AtLeastOnce*: Every message sent will be delivered or an error will be raised on at least one endpoint. Some messages may be delivered more than once.
- *ExactlyOnce*: Every message sent will be delivered without duplication or an error will be raised on at least one endpoint.

- *InOrder*: Messages will be delivered in the order that they were sent. This requires that the sequence observed by the ultimate receiver be non-decreasing, but says nothing about duplications or omissions.

The *ExactlyOnce* delivery assurance is the logical "and" of the *AtMostOnce* and *AtLeaseOnce* delivery assurances. The *InOrder* delivery assurance may be combined with any of the other delivery assurances.

To achieve the required delivery assurance guarantees, the WS-ReliableMessaging protocol adds to each message's header an identifier (in the form of a URI), a sequence number and a flag to indicate if this is the last message associated with the identifier. The messages header can also contains an acknowledgment request, which causes the receiver to send back a sequence acknowledgment that contains the ranges of sequences numbers of the messages that have been successfully received (e.g. 1...5, 7, 9…20).

Along with the delivery assurance, WS-ReliableMessaging also supports the other policies that can be associated a sequence (the name used in WS-ReliableMessaging to refer to a set of messages with have the same identifier), these policies include: sequence expiration, inactivity timeout and retransmission interval.

The WS-ReliableMessaging protocol also defines a number of types of *sequence fault*, which can be used to propagate details of faults that occur during the protocol, such as: sequence terminated, unknown sequence, invalid acknowledgment, message number rollover, last message number exceeded and sequence refused.

A.16 Universal Description, Discovery and Integration (UDDI)

The Universal Description, Discovery and Integration (UDDI) [13] project's purpose is to provide standardized methods of publishing and discovering information about Web services. The UDDI project is aiming to specify an open framework for describing service, discovering businesses, and integrate business services.

Conceptually, a business can register three types of information in a UDDI registry:

- Basic contact information and unique identifiers about the company.
- Categorisation of the Web services provided by the company.
- Behavioural description of the Web services provided by the company.

The UDDI Business Registry (UBR), also know as the "Public Cloud" has been deployed. The UBR is a decentralized registry in which the content is replicated over on all the nodes that operate the UBR.

The UDDI project has specified a number of XML schemas and programming APIs to allow publishing and discovery of information about business. The UDDI APIs are based on five primary UDDI data structures types: businessEntity, publisherAssertion, businessService, bindingTemplate and tModel.

- *businessEntity*: contains the business's basic information, including contact information categorisation, description and identifiers.
- *publisherAssertion*: is used to indicate a relationships between two businesses (*businessEntity*s). This relationship is only made public if both businesses endorse the relationship.

- *businessService*: represent services provided by the business, both Web service and manual services. A *businessService* can be associated with one or more *businessEntity*s, and a *businessEntity* can contains one or more *businessService*s.
- *bindingTemplate*: contains pointers to the technical descriptions and the access endpoint of a service. A *businessService* can contain one or more *bindingTemplate*s.
- *tModel*: contains the abstract description of a particular specification or behaviour to which the service adheres. A *bindingTemplate* can contain one or more *tModel*s.

A.17 OMG Interface Definition Language (OMG IDL)

The OMG Interface Definition Language (OMG IDL) [14] is used to describe interfaces of CORBA objects. These interface definitions specify the operations the objects is equipped to execute, the input and output parameters required, and any exceptions that may be thrown. An interface can be derived from another interface, which is called a *base* interface of the *derived* interface. The *derived* interface supports all of the operations of the *base* interface in addition to its own operations. CORBA IDL allows interface definitions to be grouped into a module definition. An example of CORBA IDL is given below:

```
module CallbackTest
{
    interface PingPong;

    typedef sequence<PingPong> PingPongSeq;

    interface PingPong
    {
        boolean oper(in long level, in PingPongSeq objects);
    };
};
```

A.18 CORBA Interface Repository (CORBA IR)

The CORBA Interface Repository (CORBA IR) [15] is the component of the ORB that provides persistent storage of interface definitions; it manages and provides access to a collection of object definitions specified in OMG IDL. Interface definitions are maintained in the Interface Repository as a set of objects that are accessible through a set of OMG IDL specified interface definitions. An interface definition contains a description of the operations it supports, including the types of the parameters, exceptions it may raise, and context information it may use.

A.19 JBoss Clustering

JBoss is an open-source J2EE application server, it currently (version 3.2.2) provides full clustering support [28][29] for stateless session beans, stateful session beans, entity beans and JNDI, and partial support for JMS and message driven beans. Replication of HTTP sessions for web applications is also available.

A cluster of JBoss server instances (nodes) is split into partitions, where a node can be a participant of multiple partitions and can join and leave partitions dynamically. The service of a particular EJB or servlet is replicated across a partition. A client, if accesses the service via RMI, can cope with failure of nodes by using a smart proxy that redirects failed invocation to other members of the partition. For HTTP clients a dispatcher can be used to manage failover between replicas.

Each type of service (i.e. stateful session beans, entity beans, …) has its own mechanisms to maintain state constituency between invocations. For stateful session beans their state is multicast after each invocation to the replicas, where as for entity beans their state is read from the database before each invocation and written back to the database after each invocation. The replication mechanisms used by JBoss can cause incorrect behaviour as well as serious performance degradation in certain circumstances.

JBoss supports the deployment of replicas of software complements across the cluster, this mechanism is called *Farming*. Farming allows a software component to be deployed on one node, and to be automatically propagated to other nodes in the cluster.

A.20 Microsoft Cluster Server (MSCS)

Microsoft Cluster Server (MSCS)[30] clusters aim to host applications that use failover to achieve high availability. When Microsoft Cluster Server (MSCS) is used in conjunction with Network Load Balancing (NLB) provide geographically dispersed server clusters, but at present this is only suitable for disaster recovery.

MSCS clusters allow that applications and services running on a server cluster to be exposed to clients as virtual servers. To clients, connecting to an application or service running as a clustered virtual server appears to be the same process as connecting to a single, physical server. In fact, any node in the cluster can host the connection to a virtual server. The client application will not know which node is actually hosting the virtual server.

MSCS architecture is based on a shared-nothing model. In the shared-nothing model, each server owns and manages its local devices. Devices common to the cluster, such as a common disk array, are selectively owned and managed by a single server at any given time.

In MSCS, resources that are related or dependent on each other are associated through resource groups. Each resource group has an associated cluster-wide policy that specifies which server the group prefers to run on, and which server the group should move to in case of a failure. Each group also has a network service name and address to enable network clients to bind to the services provided by the resource group. In the event of a failure, resource groups can be failed over or moved as atomic units from the failed node to another available node in the cluster.

The applications and service that run on top of MSCS can be either *cluster-aware* or *cluster-unaware*. Cluster-aware applications can work within the cluster environment and support cluster events; also cluster-aware applications can register with the server cluster to receive status and notification information. If an application and service is cluster-unaware, this does not mean that it can't be assigned to resource groups and can be failed over, but this can only done if application and service is "naturally failoverable".

The MSCS clusters are implemented by a set of service itself, which are composed of several functional units. These include the Node Manager, Failover Manager, Database

Manager, Global Update Manager, Checkpoint Manager, Log Manager, Event Log Replication Manager, and the Backup/Restore Manager.

The MSCS products enable server clusters to support the following resources: file and print shares, physical disks, Microsoft distributed transaction coordinator, internet information services, message queuing triggers, network addresses and names, generic services or applications and generic scripts

A.21 IBM's WebSphere Clustering

IBM's application server product WebSphere Application Server, Advance Edition [31] provides support for scalability, load-balancing and failover.

WebSphere supports a mechanism called *cloning*, is allows for the creation of multiple copies of an object such as an application server. Cloning involves taking an application server that you've set up, and creating a model based upon that setup. When you have created a model, you can then create clones of that application server. The resulting extra clones can improve the performance of the original service and be regarded as identical to the original service. Clones can be located on different machines to the original service, this allows continued service in the present of machine failure.

The model which is used to create clones is a logical representation of an application server, and has the same structure and attributes as a real application server, but it is not associated with any node.

The uses of clones does place constrains on the application being run, for example, with WebSphere V3.x, the only way to share a session between clones is to persist the session to a database.

A.22 BEA's WebLogic Clustering

BEA's application server, WebLogic Server [32], allow a cluster of application servers to appear to its clients as a single server. The cluster mechanism will ensure that connection requests can be load-balanced across the cluster and that failover can occur transparently. If a reliable, high-throughput, multicast-capable WAN connection is available between two or more LANs, the WebLogic Server cluster can be configured to span these LANs. The clustering within WebLogic Server is achieved by supporting clustering within individual support systems: Web (Servlets and JSPs), Objects (EJBs and JNDI), Messaging (JMS) and Database (JDBC).

The WebLogic Server is configurable as to how it provides clustering, for example, it support for persistence HTTP session states can be achieved by configuring the WebLogic Server between three options: in-memory replication, file system persistence or database persistence through JDBC. Each option represents a tradeoff between persistence durability and performance.