NOTE TO USERS

Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Oscar MARQUEZ-RANGEL

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Master of Computer Science

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Design and Implementation of an Agent-Based Web Service Composition Framework

A. Karmouch

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

S. Dandamudi

T. Yeap

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Design and Implementation of an Agent-Based Web Service Composition Framework

By

Oscar Márquez-Rangel

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
Department of Computer Science
School of Information Technology and Engineering
Faculty of Engineering

University of Ottawa

Your file Votre référence
ISBN: 0-494-01544-6
Our file Notre référence
ISBN: 0-494-01544-6

The undersigned recommend to the Faculty of Graduate and Postdoctoral Studies
Acceptance of the thesis

Design and Implementation of an Agent-Based Web Service Composition Framework

Submitted by Oscar Márquez-Rangel,
In partial fulfillment of the requirements for
The degree of Master of Computer Science

Thesis Supervisor

Director, SITE

University of Ottawa
Ottawa, ON, Canada, 2004

Abstract

Web service technology is making strides in the creation of distributed systems. The main objective of this technology is to enhance the interconnectivity of heterogeneous systems by using standard protocols to describe, discover, and communicate with distributed software entities across the Internet. It faces some challenges in creating a robust architecture for developing software applications. Web service composition refers to the mechanisms that promote the collaboration of individual Web services to create software applications with a functionality that is the result of the integration of the individual functionalities provided by each participating service.

Some factors to be taken into account in the implementation of Web service composition approaches are (i) the manner in which services are orchestrated, coordinated, and discovered; (ii) the scalability and flexibility levels; and (ii) the allowance of individual services to fail without affecting the complete composite service execution.

Recent Web service composition techniques provide solutions to the challenges of Web service composition. However, these solutions have the following limitations: (i) the use of the standard service registry only at development time; (ii) the necessity of service providers to host additional software modules to obtain high levels of scalability and flexibility; and (iii) the lack of mechanisms to allow composite services to be created where some individual services are allowed to fail without affecting the entire service execution.

This thesis proposes the design of a Web service composition framework in which individual Web services are represented by software agents that collaborate to create composite services. The design includes the language to define composite services, the protocol that software agents follow when executing composite services and the architecture of the proposed framework. The proposed architecture provides: (i) the incorporation of standard technologies for service discovery into a dynamic service-discovery process; (ii) high levels of scalability and flexibility without requiring the service provider to host additional modules; and (iii) the incorporation of composite services that are successfully executed even when negligible individual services fail.

The proposed framework was evaluated with different sets of users that simultaneously requested the same composite service. An improvement to the basic architecture was made to decrease the processing times when the number of users is increased.

Acknowledgements

I would like to thank my supervisor Dr. Ahmed Karmouch for his guidance, support, and encouragement throughout my research. I am very thankful to my friends and colleagues at the Multimedia and Mobile Agent Research Laboratory in the University of Ottawa for their motivation and cooperation. I wish to thank all my friends that collaborated with ideas to improve this work, especially Jennifer Templin for her invaluable help in the English revision of this thesis. I extend my thanks to Victor Ramos and Marcelino Jimenez, who, along with others, encouraged me to pursue my graduate studies. Above all, I would like to thank God, Ana (my wife), and my family for always being there for me throughout my studies, for being my source of inspiration to reach my objectives, and for loving me unconditionally.

Finally, this research could not have been possible without the financial support of CONACYT, a Mexican institution that encourages young students to pursue their graduate studies and that plays an important role in technological development in Mexico.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management System
AP	Agent Platform
API	Application Programming Interface
ASTEK	Agent-based Web Service composiTion framEworK
BPEL4WS	Business Process Execution Language for Web Services
BPML	Business Process Modelling Language
BPSS	Business Process Specification Schema
CBD	Component-Based Design
CBS	Central Bank System
CORBA	Common Object Request Broker Architecture
DAML-S	DARPA Agent Markup Language for Services
DARPA	The Defense Advanced Research Projects Agency
DCOM	Distributed Component Object Model
DF	Directory Facilitator
DOM	Document Object Model
DTD	Document Type Definition
EbXML	Electronic Business using XML
FIPA	Foundation for Intelligent Physical Agents
IDL	Interface Definition Language
J2EE	Java 2 Enterprise Edition
JAXP	Java API for XML Processing
JAXR	Java API for XML Registries
JAX-RPC	Java API for XML-Based RPC
JWSDP	Java Web Services Development Pack
MAS	Multi-Agent System
MTS	Message Transport System
OMG	Object Management Group
OOD	Object-Oriented Design
P2P	Peer-to-peer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAAJ	SOAP with Attachments API for JAVA
SAML	Security Assertion Markup Language
SAX	Simple API for XML
SBD	Service-Based Design
SLA	Service Level Agreements
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration Business Registry
UML	Unified Modelling Language
W3C	World Wide Web Consortium
WSA	Web Services Architecture
WSDL	Web Services Definition Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Chapter 1
Introduction

Different technologies have been used in the creation of distributed systems. These technologies provide mechanisms to software entities to communicate with each other within the same platform. A software entity deployed in one of these platforms has restrictions on interacting with entities deployed in different platforms. Some recent technologies that have benefited the creation of distributed systems are Web Services Architecture, software agent technology, and peer-to-peer computing.

1.1 Web Services Architecture

Web services technology enhances the interconnectivity between software entities in heterogeneous architectures by using standard technologies and Internet protocols to carry information over the Internet. Web Services Architecture (WSA) is an implementation of the Service-Oriented Architecture, in which software services are the basic computing entities that offer a specific functionality. WSA extensively uses XML technologies to define the protocols that allow interconnectivity between Web services. The use of a certain Web service is described in a Web-accessible XML document using the Web Service Definition Language (WSDL). By using the WSDL document of a specific Web service, a Web service requester can discover the type of messages and the endpoint address required to invoke the Web service. Web services communicate though SOAP (Simple Object Access Protocol) messages, which carry data of the request and response messages in the invocation of Web services. SOAP message transmission is based on standard Internet transport protocols such as HTTP and SMTP. Web services can be registered in a standard

service registry, such as the UDDI (Universal Description, Discovery and Integration) business registry, to be discovered by service requesters at development time.

1.1.1 Web service composition

Web service technology is facing some challenges in providing a widely accepted, and therefore widely used, software architecture. One of these challenges is related to Web service composition. Web service composition focuses on the mechanisms to promote the collaboration between individual Web services to create a software application with a functionality that is the result of the integration of individual functionalities. This service composition cannot only be performed statically at development time, but can also be performed dynamically at execution time.

1.1.2 Web service composition issues

Software developers face some issues when developing composite Web services. The software developer must decide whether to use a proactive or a reactive arrangement of individual Web services. A proactive service composition organizes participating services in an executable business-process representation at development time. In a reactive service composition, participating services are selected and incorporated into the composition at execution time, in accordance with the existing runtime conditions.

Other factors to consider are services choreography and discovery. Service choreography can be performed by a centralized entity or through the cooperation of distributed entities in a peer-to-peer fashion. Service discovery can be carried out by using standard (UDDI) or proprietary service repositories.

The scalability and flexibility of the service composition approach are also important factors to consider. Scalability refers to the ability of multiple Web services to be integrated

into the infrastructure with minimal effort. Flexibility refers to the ability to select, from a set of Web services that offer the same functionality, the one that best provides that functionality according to the current situation.

Software developers must also take into account that (a) involved Web services are invoked by using either request-response or one-way messages (synchronous or asynchronous calls); (b) a composite service approach can integrate composite services that act as atomic services in a broader service (recursiveness); (c) a composite service approach may require Web services to implement specifically agreed-upon service interfaces to be integrated into the composition; (d) a composite service approach can allow some individual Web services to fail within the composite-service execution; (e) a composite service approach can provide fault-recovery support to roll back faulty service executions; and (f) data and control flow can be exchanged between cooperating entities either in the same representation or in separate representations.

1.1.3 Web service composition approaches

Recent service-composition approaches address Web service composition problems and offer various solutions. For example, the Business Process Execution Language for Web Services (BPEL4WS) and its related technologies offer a proactive centralized approach that is neither scalable nor flexible. The enTish infrastructure offers a reactive approach where service choreography is performed in a peer-to-peer fashion through the cooperation of software agents, and offers a low level of scalability and a medium level of flexibility. DAML-S is an agent-based reactive approach with a medium level of scalability and a high level of flexibility in which the service choreography is performed by a centralized entity. SELF-SERV offers a reactive approach with high levels of scalability and flexibility, where the service choreography is performed through the cooperation of distributed software

entities. None of the analyzed approaches allow the creation of composite services where some individual services are allowed to fail without affecting the entire service execution, and where the data and control flow are handled in separated artifacts but in one message.

1.2 Software agent technology and peer-to-peer computing

In software agent technology, agents are the basic computing entities that can structurally represent their beliefs, roles and objectives [23]. They can cooperate and negotiate to fulfill the common objective of solving complex problems. They are also [22] autonomous (they act without the influence of external agents), reactive (they detect their environment and respond to changes in it), and proactive (they independently initiate actions that are focused on their objectives). Standard agent platforms have been created by using the Foundation for Intelligent Physical Agents (FIPA) specifications to enhance interoperability between them. FIPA agents can interact with each other, regardless of the platform they are deployed in, by using FIPA-based messages and protocols.

The peer-to-peer computing model also provides some benefits to the design of distributed systems [30]. In this computing model, software entities behave as either a client or a server, depending on the role they must play to fulfill a particular objective. The entire functionality of a peer-to-peer system is distributed among the participating entities, instead of residing in a centralized server.

1.3 Motivation

Companies can deploy their software applications through Web services, to allow others to access the functionalities that their applications offer. These functionalities can be either isolated or be part of a broader software application in which the entire functionality is an integration of individual software applications. Web service composition aims to create

mechanisms to allow software applications to be created through the collaboration of individual software applications that offer a certain functionality deployed as Web services. Although current Web-service composition approaches address challenges that Web service composition faces, the following points arise:

1. The standard UDDI service registry is not used for runtime Web service discovery. In the analyzed approaches, Web service discovery is carried out through the following methods: (a) at development time by a UDDI registry, such as BPEL4WS; (b) by a semantically-enhanced UDDI registry to allow dynamic service discovery, such as the DAML-S approach; or (c) by using non-standard service registries, such as enTish and SELF-SERV.
2. To obtain high levels of scalability and flexibility in Web service composition, Web service providers must host additional software modules. The scalability and flexibility factors of Web service composition vary from non-existent in the BPEL4WS approach to high in the SELF-SERV approach. However, with SELF-SERV, a Web service provider not only has to deploy its Web services that offer a certain functionality, but must also host a software module from the SELF-SERV architecture to be able to take part into the composition process.
3. The analyzed Web service composition approaches do not allow composite services to be created where some individual services are allowed to fail without affecting the entire service execution.

The need to address these issues is the motivation behind this thesis. The use of software-agent technology to solve these problems is another driving force in this research. The ability of software agents to cooperate in achieving common objectives can be applied to perform composite services. Single agents are in charge of providing individual

functionalities and can cooperate to create the entire functionality of a composite service. Moreover, the proactive behaviour of software agents is used to periodically search a public UDDI registry for available Web services to be integrated into the composition process and discard the Web services that are no longer available. Thus, the Web-service discovery process can be performed dynamically when a composite service is being executed.

The creation of a Web service composition framework that resolves the aforementioned issues, and the benefits offered by software agent technology in the creation of such a framework, inspire an alternative approach to Web service composition.

1.4 Thesis objectives

The objectives of this thesis are:

- To present an agent-based Web-service composition framework that uses standard Web service technologies (e.g., WSDL, SOAP and UDDI).
- To develop a mechanism to execute a composite service represented as a business process, through the collaboration of software agents.
- To develop a mechanism to dynamically incorporate Web services into the composition process without the need to host additional software modules.
- To develop a mechanism to execute composite services that allows some of their individual services to fail without affecting the composite-service execution.

1.5 Thesis contribution

This thesis presents the following results:

1. The conception and implementation of a Web service composition approach that has the following characteristics:

a) Services involved in the composition are organized into an executable business-process representation; however, the Web services that are actually invoked are selected at runtime.

b) Service choreography is performed through the cooperation of software agents.

c) Service discovery is carried out by using the agent-platform Directory Facilitator to search for software agents that provide a specific service, and by using a standard UDDI service registry to search for Web services that provide a service operation through a specific service interface.

d) The level of scalability is high. When a new Web service is going to be integrated into the framework, it must implement an agreed-upon service interface and be registered in a common UDDI service registry by following a set of standard recommendations. Service providers do not need to host additional modules.

e) The level of flexibility is high. A software agent that offers a specific service within the framework can select, from a set of Web services offering the same service through the same interface, the one that best fits the user requirements and the current intermediate data.

f) The composition process allows some services to fail without affecting the entire execution by incorporating optional-composite services.

g) Data and control flow representations are handled separately in two different artifacts. However, both are transmitted by using the same message.

2. The conception of the mechanism that allows composite-service execution through the cooperation of a set of software agents. This thesis proposes a mechanism to execute a composite service, which is represented as a business process through the collaboration of software agents. The software agents in charge of the service

execution are the following: (a) a process-broker agent, which represents a composite service within the framework; (b) a process-executor agent, which initiates the execution of an instance of a service; and (c) a set of activity-executor agents, which represent individual services and collaborate to execute the composite service.

3. The incorporation of standard technologies for Web service discovery into a dynamic service discovery process. This thesis proposes a dynamic agent-based service discovery model that builds on the agent-platform Directory Facilitator and on a UDDI service registry. Software agents that collaborate in a composite-service execution search a UDDI service registry for available Web services that offer the functionality the software agent represents. These agents select, from their list of available Web services, the one that best fits in the service composition according to current conditions. Web service providers that want to incorporate their services into the composition process must register their Web services in a common UDDI registry by following a set of standard recommendations. They do not need to host any extra modules to be integrated into the framework.
4. Incorporation of optional-composite services to be executed in the framework. This thesis proposes a method to allow the execution of optional-composite services. An optional-composite service contains individual services that are categorized as either vital or negligible. An optional-composite service is successfully executed when all of its *vital services* are successfully executed. *Negligible services*, however, are allowed to fail without affecting the entire composite service.
5. Utilization of software agents grouped into teams to decrease the response time when executing composite services within the framework. This thesis proposes using teams of activity-executor agents to decrease the average composite-service

processing time when the number of users requesting a service execution is increased. Service-execution requests are spread across the teams of activity-executor agents, thereby distributing the workload.

1.6 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 provides an overview of the Service-Oriented Architecture (SOA), Web Services Architecture (WSA), software agent technology, and peer-to-peer computing. It also discuses some challenges that WSA faces, such as service composition. Chapter 3 presents concepts related to Web service composition, including its definition and the main issues in creating it. It also presents some Web service composition approaches and a comparison of them according to how they resolve the main issues of service composition. Chapter 4 presents the design of the Agent-based Web Service composiTion framEworK (ASTEK). It describes the scenario that is used in the design and implementation of the framework, the process modelling language used to define the composite services within the framework, the data representation within the framework, and the way in which a composite service is executed through the collaboration of software agents. Chapter 5 presents ASTEK's implementation. It describes ASTEK's architecture, which contains the software entities that comprise the framework, including software tools, databases, and software agents. This chapter also presents the implementation of the software agents involved in the service composition, as well as the Web service invocation, registration, and discovery processes. Chapter 6 presents ASTEK's performance evaluation. It presents the testing scenario and the service processing time evaluation that ASTEK was subjected to. This chapter also presents an improvement upon the basic architecture to decrease the service processing times. Finally, Chapter 7 presents the conclusions of this thesis, followed by some suggestions for future work.

Chapter 2
Background

The design of distributed systems has benefited from emerging technologies that incorporate new techniques and improve upon already existing solutions. This chapter presents some recent technologies for creating distributed systems that have caused a commotion among researchers within industry and academia. These technologies offer individual solutions that can be collectively used to create more robust and reliable computing platforms for better implementation of distributed systems.

First, this chapter explains the Service-Oriented Architecture (SOA), which arose to provide interoperability among heterogeneous systems and in which services are the basic computing entities that offer a certain functionality to be used by different applications. Second, it describes the Web Service Architecture (WSA), which is an implementation of the SOA in which the Internet is used as the medium of communication. Third, it describes software agent technology, which focuses on developing systems by using software entities that can cooperate and negotiate to fulfill a common objective. Finally, it describes peer-to-peer (P2P) computing, which is a technology that uses software entities that can act as clients or servers, depending on the role they must play to meet a common objective.

2.1 Service-Oriented Architecture

The design of software systems by using a collection of individual elements that are integrated in such a way that the functionality of the whole system depends on it is not new. There has been an evolution from object-based to component-based software design to achieve the reusability of existing software objects, and from component-based to service-based software design to achieve interoperability between software components. This

section describes the use of objects, components, and services in the creation of software applications.

2.1.1 Object-oriented design

A software application can be built by using software objects that model real-world entities. The state of an object is represented by the information stored in its variables. An object performs a specific behaviour that is represented by the functions or methods used to retrieve or modify the object's state. The object's interface defines a set of public methods that an external object can interact with. Objects are controlled by external entities, either by other objects or by legacy programs. This approach brought about some benefits, including modularity, shorter development time, and information abstraction. The main weakness of the object-oriented design is that reusability can not be achieved as expected; objects written in a specific language cannot be used by applications written in a different language [1]. To resolve this inconvenience, the research community focused on the approach of the component-based system design (CBD).

2.1.2 Component-based design

Frankel [2, p.10] defines a component as a "software module that can be packaged in compiled form so that it can be [either] independently deployed as part of applications or assembled into larger components." A component can be made available through data networks in order to be used in distributed applications, regardless of the programming language it was created with. However, it can only be used within one platform. In this context, a platform is a specific combination of data-representation technology, programming language, distributed-component middleware, and messaging middleware [2].

The successful use of software components is a result of the software community's effort to agree on the creation of a standard distributed-component middleware [1]. Inside this

middleware, the Unified Modelling Language (UML) is the de facto language for describing the functionality, structure, and behaviour of components, allowing an efficient definition of the collaboration among them. Defining component interfaces is vital to component integration; therefore, an Interface Definition Language (IDL) is needed within the middleware. Also, an entity playing the role of component repository must be integrated into the component middleware to categorize and discover components. Examples of widely used distributed-component middleware include the Distributed Component Object Model (DCOM) from Microsoft, the Remote Method Invocation (RMI) from SUN Microsystems, and the Common Object Request Broker Architecture (CORBA) developed by the Object Management Group (OMG). Each infrastructure uses its own protocol for component intercommunication in such a way that the components residing in a given component middleware cannot interact with components residing in another component middleware.

Using components within specific middleware enhanced the software systems design. However, the need for interoperability among components created in different middleware arose when developers wanted an external functionality of a certain software component to be used by software components residing in another middleware.

2.1.3 Service-based design

A software service is a software entity that, in addition to the interface-based and discoverability properties of components [3], (i) exists as a single instance offering its functionalities to a variable set of clients; (ii) offers more functionality than a component-based entity; and (iii) interacts with other services and clients with loosely-coupled message-based methods using standard protocols.

The definition of service interfaces is also vital for interaction between services and clients, because the service functionality is made available through its interface definition. A

peculiar property of the Service-Based Design (SBD) is that different services can implement their functionality using a common interface, and a single service can implement its functionality using different interfaces. This property is used when services are assembled to create a complete software system. Moreover, a service is a software entity that is [4, p.19]: (i) well-defined (it offers a specific functionality); (ii) self-contained (it does not need other services to provide its functionality); and (iii) independent of the context or state of other services.

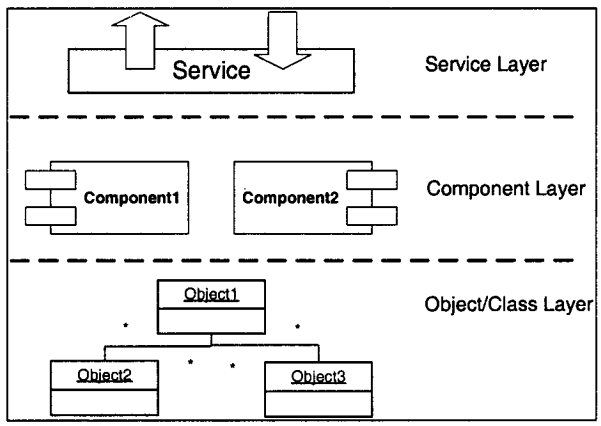


Figure 2-1. Application implementation layers [3]

Figure 2-1 depicts how a software application is implemented using a collection of software entities grouped by layers. The functionality of the application is made available through a service that uses well-defined interfaces. A service can be built on component-based approaches using existing reliable components. Components consist of objects that provide partial functionality. Note that services focus on the integration of dissimilar platforms and do not have the same features as distributed objects [5].

2.1.4 Service-Oriented Architecture

Service-Oriented Architecture (SOA) defines the entities that integrate services to create a complete system, and describes the mechanism used in such integration.

As shown in Figure 2-2, SOA consists of the following entities [6]: (i) the service provider, which provides a set of services; (ii) the service requester, which requests a certain service to fulfill a specific objective; and (iii) the service repository, which provides publishing services to providers and searching services to requesters. SOA entities are interconnected in a communication network and exchange messages by using standard protocols.

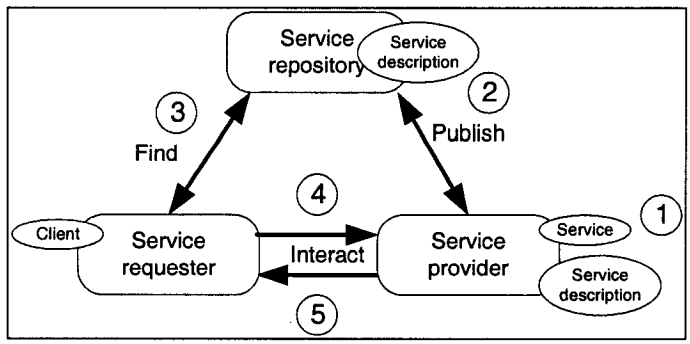


Figure 2-2. Service-Oriented Architecture (cf. Figure 1 [6])

The course of action inside SOA, as shown in Figure 2-2, consists of the following steps [6]: (1) the service provider creates the service and describes the service functionality by defining its interface and the type of messages it can process; (2) the service provider publishes its service in the service repository to be discovered by service requesters; (3) the service requester searches for services published in the service repository and finds the one that best fits its requirements; (4) once the service requester recovers the description of a service functionality from the service repository, it finds out the kind of messages it needs to invoke the service, then begins interacting with the service by sending a service invocation message to it; and (5) the service provider processes the request and returns a message to the service requester containing the service result.

In the previously described scenario, some observations arise: (a) the service requester can be either a legacy application or another service; (b) the service provider is responsible for updating the published service information whenever the service definition is modified or

the service is shut down, preventing erroneous information from being published by the service repository; (c) the service repository can be used for different purposes: for finding service providers by using keywords such as the name or category of the company; for finding services offered by a certain provider; for finding services deployed using a given interface; or for finding information related to the service usage; and (d) service requesters can search the repository for services at the time of development or execution.

2.2 Web Services Architecture

Web Services Architecture (WSA) is an implementation of Service-Oriented Architecture. The Internet is the environment where services are situated and Internet protocols are used to exchange messages among software entities. Figure 2-3 depicts WSA.

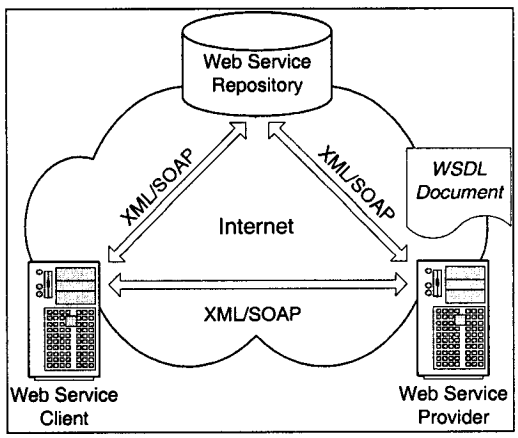


Figure 2-3. Web Services Architecture (cf. Figure 1.6 [8])

Every entity defined in SOA is implemented in WSA. Web service providers offer Web services that are made available to clients through the Internet. Communication among Web services is carried out by using a standard message format, provided by the Simple Object Access Protocol (SOAP). Web services use the Web Services Definition Language (WSDL) to create a Web-accessible document that describes how the service must be invoked. The Web service repository is implemented by either a Universal Description, Discovery and

Integration (UDDI) business registry or by an Electronic Business using XML (ebXML) registry. The Web service repository provides publishing services to Web service providers, and discovery services to Web service clients. As a result of standard protocols, Web services are not tied to specific platforms.

The rest of this section: defines Web services; describes the technologies that are used within Web Services Architecture (i.e., XML, SOAP, WSDL, and UDDI); presents some challenges that Web service technology faces; presents the Web service stack, which contains some protocols and specifications that solve some challenges that Web service technology faces; and describes two initiatives for implementing Web services platforms.

2.2.1 Web service definition

The World Wide Web Consortium (W3C) has defined a Web service as:

> "A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." [7]

The interoperability of software applications on different platforms is enhanced with Web services because they use standard communication protocols over the Internet (e.g., SOAP and HTTP), a standard language for describing the Web services (WSDL), and a public-standard service repository (UDDI or ebXML registry) to publish and search Web services. Thus, the Web, which was initially created for human-to-software interaction, is now changing to carry information for software-to-software interaction [8].

2.2.2 Extensible Markup Language

The Extensible Markup Language (XML) is a meta-language—a language that describes new languages—that consists of "a set of rules for describing structured data in plain text

rather than in proprietary binary representations" [8, p. 3]. An XML document is *well-formed* if it follows the structural rules, and is *valid* if it fits the definition of either a Document Type Definition (DTD) or an XML schema artifact.

XML has been widely accepted by the software industry, and used in a wide range of applications, but primarily for [8]: (i) data interchange: by using a previously agreed-upon data description, different companies can exchange and process data regardless of the platforms they use; (ii) software configuration: describes instructions and actions in a common representation; and (iii) protocol definition: describes how heterogeneous systems communicate with each other, i.e., the processes and messages that are involved in the communication.

2.2.3 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) is an XML-based protocol for exchanging messages between software entities using the Internet. It defines a standard message format for carrying XML data that primarily consists of the request and response messages, with details on their respective parameters required for remote-method invocation. SOAP message transmission is based on standard Internet transport protocols such as HTTP and SMTP. SOAP is based on XML-RPC, a set of specifications for invoking methods, functions or procedures across the Internet [8].

2.2.3.1 SOAP 1.1

SOAP is an XML protocol that encloses XML data inside an XML envelope to be sent anywhere across the Internet. SOAP incorporates a set of rules for creating messages to provide improved interoperability among system entities. The syntax and semantics of messages is guaranteed due to the use of namespaces and XML Schemas in such a way that messages with incorrect namespaces are discarded by SOAP applications [9].

SOAP 1.1 specification defines three concepts [9]: (i) an envelope for defining the message content; (ii) a set of encoding rules for defining a serialization mechanism to manipulate structured datatypes between partners; and (iii) an RPC representation for defining rules for representing remote-procedure calls and responses.

The envelope concept, which represents the SOAP message, is composed of three elements [9]: (i) an envelope, which is the core of the XML document and defines the entire SOAP message; (ii) an optional header, which contains information used for security, routing, or context-processing actions in the SOAP message; and (iii) a compulsory body, which contains the actual information being transported and can be either data with domain-specific XML information or a remote procedure call.

The combination of SOAP messages can represent any of the following operations between applications: (a) a one-way operation, used when the server receives a SOAP message and processes it without immediately returning a response message; (b) a notification operation, used when the server sends a message to one or more clients without a previous request message; (c) a request-response operation, used when a client sends a request message to the server, who, after processing the request, returns the corresponding response message; and (d) a solicit-response operation, used when the server sends a message to the client and receives the related response message from the client.

An example of a request-response operation is included in Appendix A, which shows both request and response SOAP messages.

2.2.3.2 SOAP 1.2

SOAP 1.2 specifications arose from the desire to improve upon SOAP 1.1 features [10]. Although SOAP 1.2 has become a W3C specification, it has yet to be fully implemented; some Web service platforms have not yet included SOAP 1.2 in their products.

2.2.4 Web Services Definition Language

The Web Services Definition Language (WSDL) is an XML-based language that describes how Web services must be invoked, and defines their interface and implementation details [11]. The interface describes the messages and parameters that a Web service can receive and respond to, and the implementation describes how messages are processed and where the Web service can be accessed. By using a WSDL document that describes a specific Web service, clients can figure out the kind of SOAP messages necessary to invoke the Web service operations.

A WSDL document has the elements shown in Figure 2-4. A WSDL document is divided into two sections [11]: the Service Interface Definition, which contains the types, message, portType and binding elements, and the Service Implementation Definition, which contains the service and port elements. These elements are presented in the following subsections.

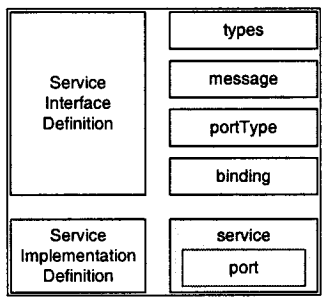


Figure 2-4. WSDL elements

2.2.4.1 Service interface definition

This section contains a description of the elements of a service interface; they are presented in an order that helps their description.

2.2.4.1.1 PortType element

The portType element defines the operations that the Web service provides. Each operation is defined by its name, the name and order of its parameters, and its input and output

messages. For example, if a Central Bank System (CBS) provides the operations described in Figure 2-5(a), the portType content looks like Figure 2-5(b), which shows only the definition of the *requestWithdrawal* operation.

```
(a) CBS operations:

Response requestDeposit     (Card card, double amount, String concept);
Response requestWithdrawal  (Card card, double amount, String concept);
Response requestUpdate      (Card card, String concept);
Response requestTransference (Card fromCard, Card toCard, double amount, String concept);
```

```
(b) WSDL entry defining the CBS System operations:

<portType name="CbsIF"> ……..
 <operation name="requestWithdrawal" parameterOrder="Card_1 double_2 String_3">
  <input message="tns:CbsIF_requestWithdrawal" />
  <output message="tns:CbsIF_requestWithdrawalResponse" />
 </operation> ……..
</portType>
```

Figure 2-5. WSDL portType element

2.2.4.1.2 Message element

The message element defines the message format. Each received or sent message must be defined by its name, and the name and type of each parameter. In the example, the requestWithdrawal operation uses two messages: a request message and a response message. Figure 2-6(a) shows the definition of these messages.

```
(a) WSDL entry defining two messages:

   <message name="CbsIF_requestWithdrawal">
      <part name="Card_1" type="tns:Card" />
      <part name="double_2" type="xsd:double" />
      <part name="String_3" type="xsd:string" />       </message>
   <message name="CbsIF_requestWithdrawalResponse">
      <part name="result" type="tns:Response" />       </message> ……..

(b) WSDL Types entry:

<types> . . .
   <complexType name="Card">
    <sequence>
      <element name="expirationDate" type="string" />
      <element name="holderName" type="string" />
      <element name="number" type="string" />
      <element name="pin" type="int" />
    </sequence>
   </complexType> . . .
</types>
```

Figure 2-6. WSDL message and types elements

2.2.4.1.3 Type element

The type element defines the complex-data types used in the messages. Figure 2-6(b) shows the definition of the complex-data type called *Card* in the example.

2.2.4.1.4 Binding element

The binding element describes the attributes used for a specific portType.

2.2.4.2 Service implementation definition

The service element contains a set of port elements. Each port element includes the endpoint address of each Web service binding where the WSDL document can be found. Figure 2-7 shows an example of the service and port elements.

```
<service name="CBSService">
   <port name="CbsIFPort" binding="tns:CbsIFBinding">
      <soap:address location="http://mmarl-11:8080/cbsservice/cbs"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" />
   </port>
</service>
```

Figure 2-7. WSDL service and port elements

2.2.5 Universal Description, Discovery and Integration business registry

The Universal Description, Discovery and Integration (UDDI) business registry is a general platform-independent repository that stores and categorizes service providers and the services they offer [12]. Because of the recent interest in Web services, UDDI has been used, with certain modifications, to describe Web services. Web service requesters use UDDI to find the services that best fulfill their requirements as well as details on how to establish communication with them. To communicate with a UDDI registry, service providers and requesters use a specific UDDI implementation that generates SOAP messages to exchange information [12].

The UDDI Project, a group of industries, governs the UDDI specifications. It has established three specifications: UDDI V1, UDDI V2, and UDDI V3. At the time of this writing, UDDI V2 is completely supported by many products. This is not the case with UDDI V3, which is in the process of being implemented. Therefore, this study refers to UDDI V2.

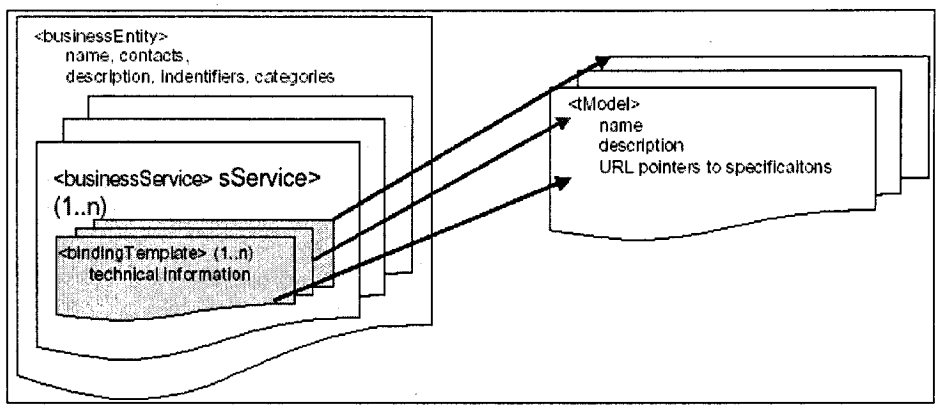


Figure 2-8. UDDI structure [12]

A UDDI entity is formed by UDDI registries. Each one contains information about the provider, services offered, service-binding details, and service specifications, all grouped in an XML document [13]. The structure of a UDDI entry is shown in Figure 2-8.

The following is a description of each component of a UDDI entry [13].

1. **BusinessEntity**: Provides information related to the company, such as the name, description, category, and contact information. This information is used for the white pages service, which lists companies according to name.
2. **BusinessService**: Provides a list of registered services offered by the company, including a short description, a category, and a set of binding-template entries that contain technical information on implementing the service. This information is used for the yellow pages service, which lists companies according to the services they offer.

3. **BindingTemplate**: Every service registered in the businessService entry has a set of bindingTemplate elements, with information about technical details for using the service. The information in the bindingTemplate entry is used for the green pages service, which lists the details on how the services can be invoked.
4. **TModel**: A particular component of every binding template, that provides a set of references to specifications for proper use of the service. A tModel may contain a pointer to technical specifications, such as a WSDL document pointer.

A technical note entitled *Using WSDL in a UDDI Registry, version 2.0* was presented by the UDDI Project to explain the relation between WSDL and UDDI, and to "describe a recommended approach to mapping WSDL descriptions to the UDDI data structures" [14]. The proposed mapping is used for enabling automatic registration of WSDL definitions in UDDI, and for accurate and flexible UDDI queries based on specific WSDL entries. Appendix B shows the mapping between WSDL and UDDI V2 data models.

2.2.6 Challenges of Web service technologies

The main objective of Web service technology is to create software applications by using existing discrete services that collaborate to provide the entire application's functionality. Therefore, these discrete services are preferably bound dynamically, rather than being selected statically. In this way, requesters could dynamically select, from among a set of services that provide the same functionality, the one that best meets their requirements. Moreover, discrete services could negotiate between each other to provide a certain functionality to the application without the intervention of external entities. This integration of and collaboration between services lead to the creation of new services from existing ones.

Some issues must be taken into account in the creation of a robust Web service technology. The following sections describe the main challenges this technology deals with to effectively achieve its objectives.

2.2.6.1 Service description

A description of services includes information not only about their functionality and interfaces but also about their non-functional characteristics and constraints, which are parameters that could be considered by other services at negotiation time [15]. A semantic description is also necessary for better service discovery.

2.2.6.2 Service composition

A service can be composed of low-level services. Both the interrelation among individual services and the manner in which they are arranged inside a process can be defined [15]. The service execution should meet the ACID transaction properties (Atomicity, Consistency, Isolation, and Durability) [34]. A service can also be defined either statically or dynamically. Moreover, a model for control and data flow can be used within the service composition.

2.2.6.3 Service delivery

A mechanism for monitoring the service invocation state at execution time is required. With such a mechanism, users and providers can react to unexpected situations such as when the service-level agreements have not been met or when services are not available [15].

2.2.6.4 Service discovery

When clients discover a set of services that meets their requirements, a collaboration phase between clients and service providers may be necessary to reach an agreement on the terms and conditions for providing the service. As a result, users can select among the discovered services the ones that best meet their requirements [15].

2.2.6.5 Security

Some security mechanisms may be added to WSA to provide accurate message exchange between partners and precise control over authentication and authorization concerns when malicious users are attempting to invoke restricted services [17].

2.2.7 Web service stack framework

M. Turner, et al., present a Web service stack framework [15] that shows how existing technologies deal with the aforementioned challenges of Web service technology (except security). Figure 2-9 depicts the Web service stack framework.

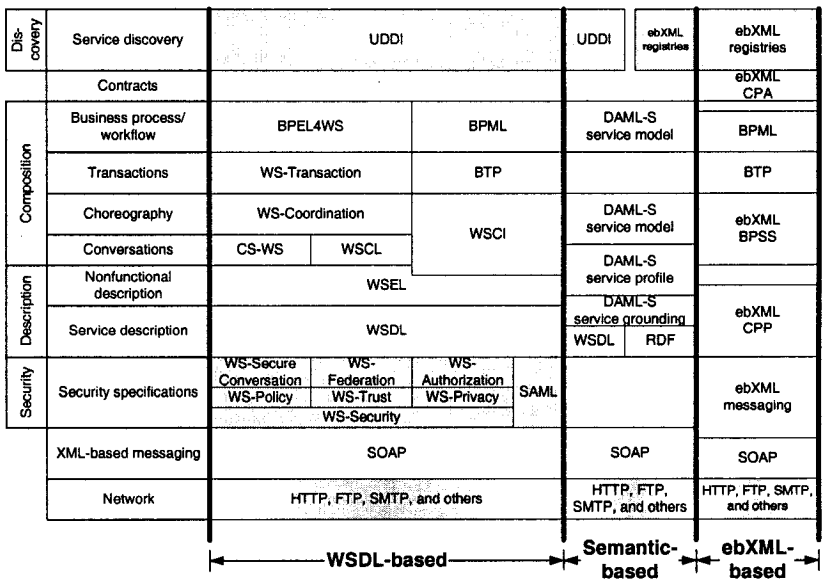


Figure 2-9. Web service stack framework (cf. Figure 2 [15])

The framework is divided into three columns [15]: WSDL-based, semantic-based, and ebXML-based groups. The WSDL-based group contains technologies that use or extend WSDL. The semantic-based group includes technologies that are based on the Semantic Web Project using the Resource Description Framework (RDF) for semantic creation and the DARPA Agent Markup Language for Services (DAML-S) for service description. Finally, the ebXML-based group includes specifications of the ebXML consortium.

The framework is also divided into the following layers [15]:

1. **Network layer:** Uses HTTP, FTP, or SMTP as transport protocols.
2. **XML-based messaging layer**: Provides a mechanism to exchange messages by using standard protocols (e.g., SOAP).
3. **Security layer:** Enables the authentication, authorization, and accurate exchange of messages to provide secure transactions within the architecture.
4. **Service description layer:** Describes the service functions using details about their interface and implementation.
5. **Non-functional description layer**: Describes the non-functional parameters used for selecting services, such as quality of service and cost.
6. **Conversations layer:** Describes the type of data contained inside exchanged messages as well as the sequence of messages in a process execution.
7. **Choreography layer:** Coordinates Web services to achieve a common goal using methods that establish the order in which the involved Web services are invoked.
8. **Transactions layer**: Monitors transactions between collaborating Web services.
9. **Business process and workflow layer:** Models the integration of individual Web services into a business process as a way of describing high-level compositions of services, by using specifications to express the data and control flow within the process execution.
10. **Contracts layer:** Describes protocols for automatic service-based negotiations where contracts are used to describe the service-level agreements between partners.
11. **Discovery layer:** Enables service providers to publish their services, and clients to search for and discover services.

2.2.8 Solutions to challenges that Web service technology faces

This section explains how current Web service technologies overcome the challenges that Web service technology faces.

2.2.8.1 Service description

Problems associated with service description are handled by the WSDL-based technologies with the Web Services Endpoint Language (WSEL), an IBM specification that aims to describe the non-functional and negotiable parameters of Web services. Semantic-based technologies introduce the DAML-S service profile as a method of describing both functional and non-functional features. EbXML-based technologies deal with this issue by incorporating the Collaboration Protocol Profile (CPP), which provides greater functionality for service description and error handling than WSDL. [15]

2.2.8.2 Service composition

The WSDL-based technologies address service-composition problems by introducing the Web Services Conversation Language (WSCL) and the Conversation Support for Web Services (CS-WS) specification. Both solutions describe the type of data and the sequence of messages exchanged between Web services. The WS-Coordination Protocol and the Web Services Choreography Interface (WSCI) are specifications for describing how composite services are connected. Transactions are monitored and managed by the Business Transaction Protocol (BTP) and by WS-Transaction specifications. The control and data-flow modelling can be expressed by using either the Business Process Modelling Language (BPML) or the Business Process Execution Language for Web Services (BPEL4WS). [15]

In the semantic-based technologies, DAML-S provides a service profile and a service model for service-composition support. EbXML-based technologies use the Business

Process Specification Schema (BPSS) along with BPML and BTP specifications to describe the internal aspects of workflow modelling and transaction processing.

Chapter 3 presents the enTish and SELF-SERV infrastructures, which, in addition to BPEL4WS and DAML-S, provide solutions for problems related to service composition.

2.2.8.3 Service delivery

WSDL-based technologies offer a solution to service-delivery related problems by using the Web Services Choreography Interface (WSCI) to describe the actions that a service performs in exceptional situations. The DAML-S service-grounding model (see Figure 2-9) solves service-delivery problems by using semantic-based technologies. EbXML-based technologies offer a third solution using the Collaboration Protocol Agreement (CPA) to describe the service-level agreements between partners; the invocation of services is based on the fulfillment of these agreements. [15]

2.2.8.4 Service discovery

UDDI cannot semantically describe services for better automatic processing of service discovery; it is limited to basic queries [15]. A DARPA project [16] presents a mapping between the DAML-S service profile and the UDDI data model, as well as some modifications to the UDDI registry to improve the semantic representation of services and enhance the service-discovery process. The ebXML registry provides better searching capabilities but it still does not have semantic specifications.

2.2.8.5 Security

IBM and Microsoft have joined efforts to create specifications that address security issues [17]. The basis of these specifications is WS-Security, which is a model for accurate exchanging of SOAP messages that incorporates signature and encryption headers, as well

as security tokens, into the messages. WS-Policy, WS-Trust, and WS-Privacy are based on WS-Security. WS-Policy describes security policies when accessing Web service endpoints; WS-Trust creates trust models for Web service interoperability; and WS-Privacy enables both clients and providers to create privacy preferences. In addition to these technologies, IBM and Microsoft recommend: (i) incorporating the WS-SecureConversation specification, which is in charge of authentication issues and the establishment of secure conversation sessions; (ii) the WS-Federation specification, which manages trust relations within federated environments; and (iii) WS-Authorization, which focuses on data authorization and the policies to grant it. The OASIS consortium offers different solutions to the security issue. It is working with the Security Assertion Markup Language (SAML), which is a standard for exchanging messages that contain authentication and authorization information as part of a SOAP message [18]. EbXML-based technologies use the ebXML messaging specification to incorporate authentication and contextual information within the header of SOAP messages.

2.2.9 Web services implementation

Two Web service initiatives have been competing for the implementation of Web service platforms: the .NET framework and the Java 2 Enterprise Edition (J2EE) standard. The .NET framework is a Microsoft initiative that only works on Microsoft Windows operating systems. The J2EE standard, with more than 30 implementations, is an initiative agreed upon by a set of companies [19] that ensures the implementation of Web services regardless the platform being used. SUN, IBM, Oracle, HP, and BEA have their own J2EE implementation.

SUN's implementation of J2EE will be used in this study, given the advantages that it has over .NET. These advantages are the following: (a) the design of J2EE implementations is

based on Java language; therefore, external Java software tools can easily be integrated into the development of Web services; (b) J2EE implementations can be installed on the main operating systems (e.g., Windows, Linux, or Unix); (c) SUN provides an implementation of the J2EE specifications free of charge; (d) SUN provides invaluable forums for solving problems that developers may encounter when developing J2EE applications; and (e) SUN fully implements the J2EE specification.

2.3 Software agent technology

Software agents emerged from the evolution of software objects [20]. In addition to the properties of software objects, software agents can structurally represent beliefs, roles and objectives [23]. Communication among agents is enhanced by the use of high-level and well-expressed languages. Software-agent technology incorporates into software development the concepts of cooperation and negotiation among software entities, as well as the concept of computational communities [21].

2.3.1 Software agent definition

Software agents are software entities situated within an environment and have the following properties [21]: (1) **autonomy:** they use their own protected internal state to freely act without the influence of external entities; (2) **reactivity:** they detect the environment and respond to changes in it; (3) **pro-activity:** they can independently initiate actions that are focused on the agent's objectives; and (4) **social ability**: they can engage in social activities to cooperate or negotiate to complete a collective task. Moreover, agents are [22] (1) **mobile**: they can move across agent platforms; (2) **truthful:** they only share accurate information; (3) **benevolent**: they only do what they are requested to do; and (4) **rational**: they are focused on meeting their objectives.

2.3.2 Agent platforms

Agents are deployed within an environment, called an Agent Platform (AP), that provides them with the necessary facilities to meet their objectives. Perhaps, the main function that agent platforms must provide to agents is the ability to communicate with each other. This allows them to interact with their peers. [23]

Several agent platforms have been implemented for creating software applications in which software agents are the main software entities. Early agent platforms were implemented with proprietary technologies for deploying software agents and for enabling communication among them. During the development of new agent platforms, the problem of interoperability arose: agents deployed within an agent platform could not interact with agents residing in a different agent platform. Therefore, standards were clearly necessary in the creation of agent platforms to guarantee interoperability.

In 1997, the Foundation for Intelligent Physical Agents (FIPA) [24] was formed as a result of the collaboration among companies and universities involved in software agent technology. The main objective of FIPA is to develop specifications to provide interoperability among agent platforms. Several organizations have taken into account the FIPA specifications in the creation of agent platforms. Some examples [25] of these platforms are: ZEUS, from BTexact Technologies (UK); AAP, from Fujitsu Labs (USA); JADE from CSELT (Italy); and FIPA-OS from Nortel Networks (UK).

2.3.3 FIPA agent platform

FIPA describes the standard infrastructure where FIPA agents are deployed [24]. This infrastructure is represented in the Agent Management Reference Model (AMRM), which is shown in Figure 2-10. The AMRM defines an Agent Platform (AP) as the "physical infrastructure" where agents are located [24]. An AP is deployed in one or more computers

that can have different operating systems, and is composed of [24]: (a) a mandatory and unique Agent Management System (AMS); (b) a Message Transport System (MTS); and (c) an optional Directory Facilitator (DF). All the components within the AP are agents. They can communicate with external software entities to integrate them into the agent technology, or provide new services within the Agent Platform by using the external software entities.

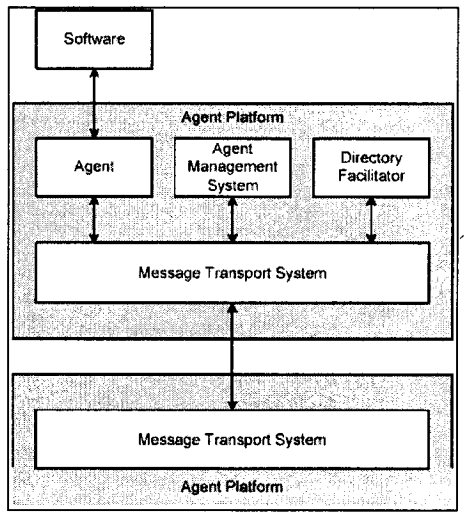


Figure 2-10. Agent Management Reference Model [24]

2.3.3.1 Agent Management System

The AMS is the core of the AP [24]. It provides management functions for all of the components required to operate the AP; thus, the AMS is in charge of the AP's operability. The AMS assigns a unique Agent Identifier (AID) to every agent that is created in the platform. The AMS contains a list of all existing agents within the platform that is used by agents to search for other agents. The AMS is also used to register new agents (either agents created inside the platform or agents that are moving in from another platform); to deregister agents (when an agent dies or migrates to another platform); to modify agent descriptions (when necessary); to search for information about certain agents; and to retrieve the description of the platform. Moreover, the AMS can create, suspend, terminate,

resume, and invoke an agent, as well as manage which resources each agent is allowed to use.

2.3.3.2 Message Transport System

The MTS is responsible for message exchanges among agents residing within the same AP, and with agents belonging to external platforms. [24]

2.3.3.3 Director Facilitator

Agents use the DF to publish their offered services and to search for agents offering a certain service. Agents are responsible for the accuracy of their published information; therefore, an agent must modify its published information every time it changes the way in which other agents must invoke its services, and erase its information when the agent completes its execution. [24]

2.3.4 FIPA agent communication

Agents are able to communicate with each other only if they share the same ontology and content language. An ontology is a vocabulary that represents the shared knowledge and defines the concepts and their relations in a specific area of interest [26]. A content language is the language used to represent the beliefs, desires, or intentions [27] that an agent wants to communicate. Thus, agent communities are created by grouping agents that share the same kind of knowledge and perform certain functions in the same application area. Therefore, agents can simultaneously belong to different agent communities.

The Agent Communication Language (ACL) is the language that enables agents to communicate. FIPA ACL provides a set of language primitives to exchange information among agents, regardless of the ontology or content language they use.

FIPA agents exchange FIPA ACL messages to communicate their desires, beliefs, or intentions. [24] These concepts can be expressed by using a standard set of FIPA communicative acts, which are words that express the meaning associated with the message content. Some of these communicative acts are: *inform*, *request*, *agree*, *failure*, *query-if*, *propose*, and *non-understood*. For example, when an agent cannot decipher the content of a received message, it can return a message with *non-understood* as a communicative act.

A FIPA ACL message is composed of a set of message parameters [28]. The crucial parameters that a FIPA ACL message contains are the following: *performative,* which is compulsory and expresses the communicative act of the message; *sender*, which indicates the identity of the agent that sent the message (which may not be present if the sender decides to remain anonymous); *receiver*, which is compulsory and indicates the identity of the intended receiving agent; *content*, which is a sentence that represents the actual intention or goal of the message; *language*, which indicates the language used to express the message content; and *ontology*, which represents the ontology used to structure the sentence content.

FIPA defines a set of interaction protocols, which represent specific sequences of messages exchanged among agents in typical conversations [29]. Interaction protocols are reliable patterns for interaction among agents that could be used to enhance the development of agent-based systems. Some of these protocols are the *Request Interaction Protocol*, which is used when one agent requests another to execute a certain action; the *Query Interaction Protocol*, which is used when an agent asks another for the value of a specific proposition, or when requesting a specific object; and the *ContractNet Protocol*, which is used when one agent negotiates the execution of a certain task with a group of

agents to select the agent or agents that best execute the task. Figure 2-11 depicts the Request Interaction Protocol.

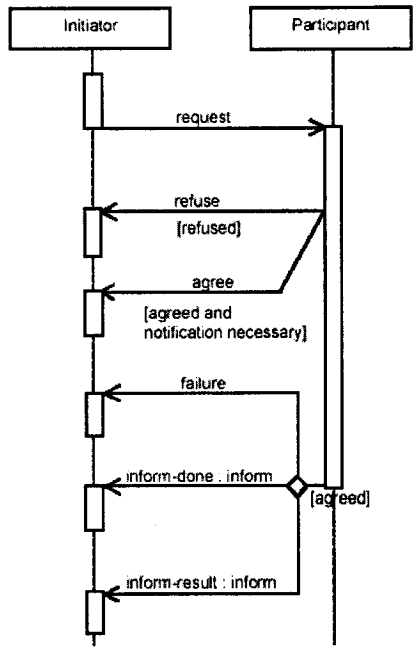


Figure 2-11. Request Interaction Protocol [29]

Interoperability among FIPA agents is guaranteed due to the standard FIPA ACL and the common ontologies and content languages. As a result, FIPA agents can communicate with each other regardless of the FIPA-based platform in which they are deployed.

2.4 Peer-to-peer computing

Peer-to-peer computing is a computing model in which software entities, called peers, behave either as a client or server depending on the role they must play to fulfill a certain objective [30]. In a peer-to-peer infrastructure, the entire functionality of the system is dispersed among the participating entities, instead of residing in a centralized server, as is the case in the client-server model. Peers need to advertise their offered services in order to be located by other peers and interact with other entities. Some applications of this model

include file transference, instant messaging, storage schemas, and collaborative systems. In this last application, the functionality of the entire system is provided by the coordinated interaction of the component peer entities.

The creation of peer-to-peer systems over the Internet requires the development of one overlay structure in which peers can interact with each other. The creation of this overlay structure is enhanced by the use of Web Services Architecture, which enables peer entities to advertise their functionalities in order to be located by other peers and provide their functionality using standard definitions of interfaces and message representations [30].

2.5 Summary

In this chapter, some recent technologies that have enhanced the design of distributed systems were presented, namely: Service-Oriented Architecture (SOA), Web Services Architecture (WSA), software agent technology, and peer-to-peer computing. SOA provides an infrastructure where software applications can be built by using discrete software components regardless of the platform these components are deployed in. Thus, SOA provides the desired interoperability among heterogeneous platforms. Within SOA, a software service represents the functionality a provider offers. This functionality is defined by the service interface that contains the allowed mechanism to communicate with the service. Services are made public in a service repository. Users can access a service repository to search for available services that meet certain criteria. Services interact with each other to build the whole functionality of the system by using standard protocols for message exchange.

Web Services Architecture maps the existing components in the SOA onto Web technologies. Web services are described by using the standard Web Services Definition Language (WSDL) and advertised by using either the UDDI or the ebXML service registries.

Once clients know how the Web service must be invoked, they interact with it by using SOAP as the standard message format. New challenges arise in the creation of applications using WSA. First, the service description may not only consider the Web service functionality and its interface definition, but may also semantically describe the negotiable parameters to improve service discovery and composition. Second, a dynamic mechanism that enables collaboration among individual services is necessary to reach a common objective. Third, monitoring the state of service invocations and reacting to outstanding situations are necessary. Finally, a negotiation among different service providers is necessary at discovery time so that they agree on the conditions of the services being provided. Some proposals that deal with these challenges were presented in this chapter.

Software agent technology was developed to improve upon object-oriented technology by adding to software objects the ability to structurally represent beliefs, roles and objectives. It also improves communication among software entities by using high-level and well-expressed languages. The Foundation for Intelligent Physical Agents (FIPA) is focused on creating standards for interoperable Agent Platforms. FIPA provides agents with the ability to communicate and cooperate regardless of the FIPA platform in which they are deployed. FIPA standards also define some general interaction protocols for representing patterns for agent communication.

Peer-to-peer computing and Web services have strongly influenced the creation of distributed computing. They enable the Internet to create distributed systems to solve specific problems regardless of the platform or location of their software components.

Chapter 3
Web Service Composition

Web service technology is facing some challenges in how it provides a more robust infrastructure for deploying Web services. The success of Web service technology depends on how these challenges are overcome. Web service composition is a challenge for providing mechanisms to promote the collaboration of individual Web services to create software applications [31]. The entire functionality of these applications is created by integrating the individual functionalities of each Web service.

This chapter covers the definition of Web service composition as well as the main issues that software developers face when integrating Web services. Moreover, this chapter presents the description and comparison of recent Web service composition approaches.

3.1 Definition of Web service composition

By deploying a Web service, a service provider can offer a functionality generated by the interaction of its internal software applications. This functionality can be made public either within its internal network, or through the Internet.

Individual Web services improve upon the distributed-system design in that they can be invoked regardless of the platform used by the involved parties. However, the real benefit of Web services is the possibility of integrate individual Web-service functionalities to create software applications with a more solid functionality [31]. Perhaps the most widely used example of such Web-service integration is a travel service system, which integrates services such as flight and hotel booking, car rental, and travel insurance. Every service provides specific functionalities within the travel service system. However, to provide an entire travel

service system, they must be invoked under certain conditions. For example, the hotel booking service cannot be invoked without the prior execution of the flight booking service.

Web service composition refers to the methods or mechanisms used to integrate individual Web services in the creation of more customized applications. It includes specifications for the proper integration of individual Web services, also known as orchestration, as well as for their execution control or coordination, also known as choreography. [31]

Individual Web services are commonly called *atomic services* and the resulting assembled Web service is called a *composite service* [38]. Thus, a composite service is a collection of atomic services—or more specifically, service operations—arranged in such a way that their collaboration carries out user requests [38].

3.2 Main issues in Web service composition

Some factors must be taken into account in the implementation of Web service compositions. These factors are how the services are orchestrated, coordinated, and discovered. When Web services are being integrated, the following factors must also be considered: scalability and flexibility; the communication mechanism used between Web services; the ability of composite Web services to take part in a broader composition (i.e., recursiveness); the definition of Web-service interfaces; the level of participation between atomic services within a composite service; fault-recovery support; and the representation of data and control flow. These factors are described in the following sections and will be used in subsequent sections to describe and compare different Web service composition techniques.

3.2.1 Service orchestration

Service orchestration refers to how atomic services are arranged to provide the functionality of the composite service [32]. There are two main methods to service orchestration: one in

which atomic services are arranged in an executable process representation before being invoked, called a *proactive, imperative*, or *procedural*, and another in which atomic services are grouped at runtime, called a *reactive* or *declarative*.

3.2.1.1 Proactive service orchestration

The proactive service orchestration uses an executable business-process representation to express how the atomic services are organized. This is useful when developing composite services that are frequently used by many users, and when atomic services are stable.

3.2.1.2 Reactive service orchestration

In the reactive service orchestration method, atomic services are selected only when they are needed at execution time, and in consideration of the user request and the existing runtime conditions. This is useful when atomic services are likely to undergo changes and when composite services are not frequently used [31].

3.2.2 Service choreography

Service choreography refers to [32] the execution-control mechanism used in the invocation of atomic services that results in a composite service. Atomic services can be choreographed either by a centralized entity or through the cooperation of distributed entities, in a peer-to-peer fashion. Moreover, because the availability of atomic services depends on many factors, such as the network's reliability or the conditions of the computer that hosts the services, a fault-recovery mechanism must be integrated into the composition approach to detect unsuccessful service invocations and react to failures according to predefined actions [33].

3.2.3 Service discovery

Service discovery is the process in which available atomic services can be located to be assembled into a composite service [31]. Service discovery can be implemented in a

centralized manner, by using service repositories such as UDDI, or in a distributed manner, by using service discovery techniques through entities distributed over a certain environment. When using a centralized repository, the service provider is required to update the information of its Web services when they are off-line or when their implementation details have changed.

3.2.4 Level of scalability and flexibility

Service orchestration approaches should take into account the properties of scalability and flexibility [47]. A Web service composition approach is *scalable* if it permits the integration of multiple Web services into the infrastructure with minimal effort. A Web service composition approach is *flexible* if it can select, from among atomic services that offer the same functionality, the one that best provides that functionality. Scalability and flexibility can be improved by defining the control logic within the composite service without considering the actual atomic services being invoked [47]. The level of scalability and flexibility can be categorized as *high, medium, low, or nonexistent.*

3.2.5 Service communication

Web services can be accessed by sending them messages and by receiving the corresponding response message (if it exists). Communication with a Web service can be either through request-response messages or through one-way messages [40].

3.2.5.1 Communication with request-response messages

In a communication with request-response messages, the user sends a request message to the service, which processes the request and immediately returns the corresponding response message. During the invocation time, the user is blocked until the response message arrives.

3.2.5.2 Communication with one-way messages

In a communication with one-way messages, the user sends a one-way request message to the service without waiting for the service response. The service would either immediately process the request, or queue the received message for future processing. If there is a response, the server sends the corresponding one-way response message to the user.

Service composition approaches must consider the type of communication that atomic services are able to use. It would be difficult to integrate atomic services that communicate in a one-way fashion within a process that requires an immediate response to requests.

3.2.6 Recursiveness

This factor is related to the ability of composite services to act as atomic services within a broader composite service [40]. Recursiveness is useful when a certain functionality cannot be provided by an atomic service, but can be carried out by the collaboration of a set of them. This collaboration should be dynamically created, without the intervention of external entities, and according to the current conditions. To be used as an atomic service, a composite Web service must be made public using its own WSDL interface.

3.2.7 Definition of interfaces

When composing services, another factor to be considered is the definition of interfaces for providing the service functionality. A service interface is the set of operations that a service offers to external software entities. The service interface can enhance its incorporation into a composite service, but can also make it more difficult to incorporate into a different one. The kind of composite service where services can collaborate must be considered when designing service interfaces. It is also useful to consider implementing one service that provides its functionality through more than one interface, as shown in Figure 3-1(a).

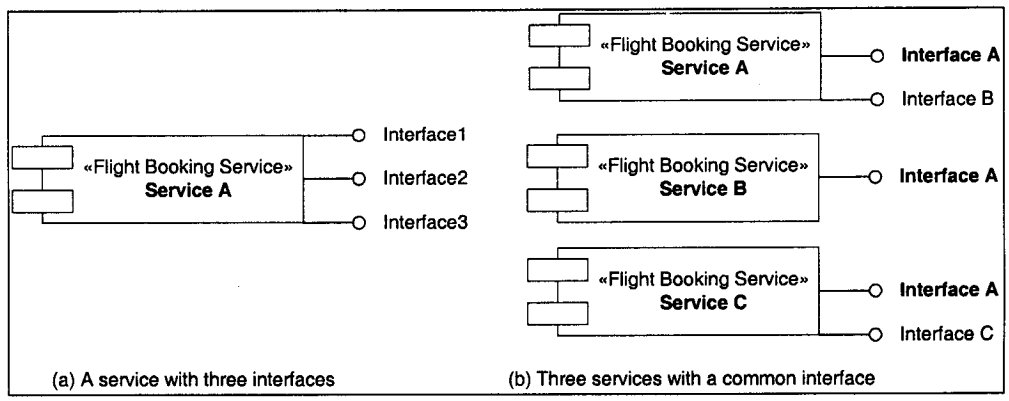


Figure 3-1. Service interface definition

The definition of interfaces also allows Web services that are deployed by different providers to offer the same functionality (e.g., a set of companies can provide the same flight booking service). This makes it easy to replace a given atomic service when the existence of a similar atomic service offering better service conditions is detected. For a better selection of atomic services, different companies can offer the same service functionality through previously agreed-upon interfaces. Figure 3-1(b) shows how three separate companies offer the flight booking service using the common *Interface A*. Thus, any of the flight booking services offered by the three companies can easily be incorporated into a composite service requiring a flight booking service with the specific *Interface A*. The selection criteria among the three services can take into account parameters such as cost, and quality of service.

3.2.8 Participation of atomic services

An atomic service can participate within a service composition as either a *vital* service or *negligible* service. It is a vital service if its execution is essential for the full execution of the composite service. However, it is a negligible service if the entire service can be executed without it. Chakraborty and Joshi [31] characterize composite services as *mandatory-composite* and *optional-composite* services based on the participation of their atomic services.

3.2.8.1 Mandatory-composite service

A mandatory-composite service is a composite service in which all of its atomic services are vital services. The successful execution of a mandatory-composite service depends on the successful execution of each one of its atomic services.

3.2.8.2 Optional-composite service

An optional-composite service is a composite service that contains vital and negligible atomic services, and its successful execution does not necessarily require the execution of all of its atomic services. During the execution of an optional-composite service, the execution of negligible services is allowed to fail without affecting the functionality of the composite service.

3.2.9 Fault-recovery support

In the development of composite services, the fact that atomic services can fail anytime must be considered [31]. If a vital atomic service fails during execution, the entire composite service also fails. A mechanism for cancelling the already executed services is needed to roll back the whole composite service execution. For instance, in the travel service scenario, the hotel booking service is a vital atomic service that is executed after the flight booking service, also a vital atomic service. If the hotel booking service fails, then the entire composite service fails; all of the already executed services must be cancelled. Thus, a flight booking cancellation operation is necessary to successfully cancel the composite service.

Treating a composite service as a transaction promotes the accurate execution of the composite service to obtain reliable results [33]. A transaction is a collection of operations that are executed sequentially to modify certain system data "from one consistent state to another" and it is considered as a single execution for the end user [34, p. 470]. The operations of a transaction are "recoverable objects" in that they can reverse the modification of the data after the transaction has failed. Thus, if the execution of a composite service fails, i.e., if one of its

vital atomic services fails or the complete execution is aborted by the user, all of its atomic services must execute their undo operation to restore the whole system to its original state. Transactions should meet the ACID (Atomicity, Consistency, Isolation, and Durability) properties, to ensure their correct and reliable execution [34].

3.2.10 Data representation

Another factor to consider in the integration of Web services is how data are represented. During the execution of a composite service, the invocation of some atomic services generates certain intermediate data that is used as input data for the invocation of subsequent atomic services participating in the same composition. A data representation is necessary to facilitate the invocation of subsequent services. Moreover, after the execution is completed, a data representation is needed to show or store the resulting information. The data representation can be included within the control flow representation, or can be handled separately.

3.3 Web service composition approaches

Some Web service composition approaches have emerged to solve issues associated with Web service composition. This section covers the most significant solutions related to this thesis. The main issues in the creation of Web service compositions that were pointed out in the previous section are the parameters that describe and compare every approach.

The Web service composition approaches that are analyzed in this section are: the Business Process Execution Language for Web Services (BPEL4WS) and its related technologies, the enTish infrastructure, the DAML-S language, and the Self-Serv framework. These approaches are described in the following subsections.

3.3.1 BPEL4WS and related technologies

The Business Process Execution Language for Web Services (BPEL4WS) is a set of specifications that models business processes based on Web services [35]. A business process describes how Web services are integrated into a single execution to implement the functionality that the process represents. BPEL4WS includes the best features of two early works in business process design: XLANG, which offers structural constructs to support sequential, parallel, and conditional process flows; and the Web Services Flow Language (WSFL) which offers graph-oriented process modelling support [35].

BPEL4WS provides an approach for modelling the behaviour of Web services when they are incorporated into a business process [32]. Developers can use BPEL4WS to model a process by describing it in an XML document. This XML document describes the control logic that governs the Web service operations involved in the process and the data dependencies on these Web services. In turn, an "orchestration engine" takes this XML document and executes it by following the control logic in which Web service operations are assembled, and by executing compensation tasks when faults occur in the process execution [36].

BPEL4WS models two different types of business processes: *executable processes* and *abstract processes*. An executable process models the orchestration of the composite service by grouping the participant processes inside a workflow. An abstract process models the choreography of Web services by specifying the messages exchanged among the Web services involved in the business process.

As depicted in Figure 2-9, BPEL4WS is a specification based on WSDL. It identifies the Web services being invoked within a business process. BPEL4WS has to be with other specifications in order to implement a complete Web service composition approach. WS-Coordination and WS-Transaction accompany BPEL4WS. WS-Coordination describes

how Web services collaborate by playing specific roles within the process execution. WS-Transaction incorporates transactional semantics into coordinated activities [38].

BPEL4WS competes with alternative specifications. The Business Process Management Language (BPML) [37] is also a language for describing business processes that can express "concurrent, repeating, and dynamic tasks". Like BPEL4WS, BPML is based on WSDL. Although developers can alternatively use both language specifications, it seems that BPEL4WS will be more widely used [15].

The Business Transaction Protocol (BTP) provides the same functionalities as WS-Transaction (the management of business transactions and fault recovery mechanisms). Whereas the Web Service Choreography Interface (WSCI) specification is the foundation of BPML for expressing the collaboration of participant Web services within the business process, WS-Coordination is the foundation of BPEL4WS [15].

As for ebXML technologies, BPML and BTP are also used for business-process modelling and transaction support. The choreography governing Web service interactions are achieved by the Business Process Specification Schema (BPSS) [15].

3.3.1.1 BPEL4WS and Web service composition issues

This subsection presents how BPEL4WS and its related technologies solve the previously discussed problems associated with Web service composition.

3.3.1.1.1 Service orchestration

BPEL4WS uses proactive service orchestration in which atomic services are statically arranged by using a business process, which represents a composite service, before being invoked. The structure of a BPEL4WS business process, in its version 1.1, is defined according to the XML-defined structure [35] shown in Figure 3-2.

```
<process …>
   <partnerLinks> … </partnerLinks >
   <variables> … </variables>
   <correlationSets> … </correlationSets>
   <faultHandlers> … </ faultHandlers >
   <compensationHandler> … </ compensationHandler>
   <eventHandlers> … </eventHandlers>
   (activities)*
</process>
```

Figure 3-2. Structure of a BPEL4WS process

The following elements define a business process:

1. ***<partnerLinks>*** describes the atomic Web services with which the process interacts. This element contains one ***<partnerLink>*** element for each partnership in the process.
2. ***<variables>*** declares the variables that represent the process state. Information, such as the intermediate data used in the business logic, is stored in here.
3. ***<correlationSets>*** enables asynchronous interactions within the process. A correlation set provides the process-execution engine with a mechanism to match receiving asynchronous messages with their intended business process instances to ensure that messages are delivered to the correct business process.
4. ***<faultHandlers>*** describes the activities that are executed when errors or exceptions occur either inside or outside the process-execution engine.
5. ***<compensationHandler>*** describes the activities to be executed when a transaction rollback occurs. A process definition considers the operation to be invoked, and determines the actions to follow when a rollback is necessary due to a failed process.
6. ***<eventHandlers>*** describes how the process deals with external events.

The BPEL4WS business process performs activities that are defined following the ***<eventHandlers>*** construct. An activity can represent either the workflow logic within the process or an action to be performed according to the workflow logic.

The following structures define the control flow within the process execution:

1. ***<sequence>*** defines a set of activities to be performed sequentially.
2. ***<flow>*** defines a set of activities to be performed concurrently.
3. ***<switch>*** allows the process to select only one activity from a set of choices based on a certain condition.
4. ***<while>*** allows the repeated execution of one activity while a certain condition is considered true.
5. ***<pick>*** allows the process to perform execution blocking while awaiting the occurrence of an external event and then execute a certain defined activity.
6. ***<throw>*** contains a set of actions to be executed when a fault occurs in the internal execution of the process.
7. ***<wait>*** allows the process to wait for a specified period of time.
8. ***<empty>*** is used to insert a *no operation* when synchronizing concurrent activities.
9. ***<scope>*** defines a *nested activity* containing its own <variables>, <faultHandlers>, and <compensationHandler> elements.

The following elements define the actions to be performed [35]:

1. ***<invoke>*** allows the invocation of a specific Web-service operation.
2. ***<receive>*** allows the process to wait for the reception of an invocation message.
3. ***<reply>*** allows the process to reply to a message received via a <receive> structure.
4. ***<assign>*** is used to allocate new values to variables.

The definition of the initial activity in the process starts with either a ***<sequence>*** or a ***<flow>*** element. Logical constructions that represent the *then-if-else* or *until* structures are implemented by combining the basic control structures.

3.3.1.1.2 Service choreography

When using BPEL4WS, service choreography is implemented by a centralized entity. A service-choreography engine, such as the BPEL4WS Java Runtime platform (BPWS4J) from IBM, is used to execute a process in a centralized manner. BPWS4J receives a workflow description and executes it in a centralized way [36]. BPWS4J ensures that atomic-service operations are executed according to the process description. BPWS4J is also in charge of executing the actions when faults arise within the process.

3.3.1.1.3 Service discovery

In the BPEL4WS approach, atomic services are integrated into the process at development time. Atomic services are provided by specific Web services that the developer uses when developing the process. There is no automatic service discovery at execution time because all the involved services are integrated into the business process description.

Developers can create business processes where some atomic services are provided by a partner service provider. In such case, the partner provides developers with the details of the Web-service interfaces. At development time, the developer can still use a service registry to search for Web-service providers, the services they offer and for invocation details.

3.3.1.1.4 Level of scalability and flexibility

BPEL4WS is neither scalable nor flexible. Whenever a Web service provided by a new business partner needs to be incorporated into the composite service, the information about how to invoke the service, such as its endpoint address, must be integrated in the business process representation at development time. In addition, the business process refers to existing and non-interchangeable Web services. The service choreography engine is not free to select a service to be invoked from a set of services that provide the same functionality.

3.3.1.1.5 Service communication

BPEL4WS supports both request-response and one-way messages to enable communication between atomic Web services and the process executor. The request-response communication is carried out by a combination of ***<receive>*** and ***<reply>*** elements within the process. The ***<invoke>*** element is used to invoke services with both kinds of messages.

3.3.1.1.6 Recursiveness

A BPEL4WS process can act as an atomic service in a broader composite service [35].

3.3.1.1.7 Definition of Interfaces

The definition of interfaces for dynamic service discovery is not applicable in BPEL4WS. In BPEL4WS, the composition of atomic Web services is carried out statically by using previously agreed-upon interfaces. A BPEL4WS process is based on the WSDL document description of each atomic service. An atomic service can provide more than one interface to offer a specific functionality, but within the process definition only one interface is used. Moreover, because there is no service discovery at execution time, there is no need to provide more than one interface for a specific functionality. Thus, the definition of more than one interface for a specific functionality is not applicable when using BPEL4WS.

3.3.1.1.8 Participation of atomic services

BPEL4WS does not support negligible atomic services. Therefore, a composite service in this approach *cannot* be an optional-composite service, and *must* be a mandatory-composite one.

3.3.1.1.9 Fault-recovery support

BPEL4WS provides fault-recovery support. The use of the ***<faultHandlers>*** and ***<compensationHandlers>*** elements makes it possible to execute activities after the detection of internal or external failures.

3.3.1.1.10 Data representation

A BPEL4WS process contains the control flow and data within the same document. At execution time, the execution engine follows the control flow described in the process and makes decisions based on the data stored in the ***<variables>*** element of the process definition. The contents of received messages are also stored in the ***<variables>*** element.

3.3.2 The enTish infrastructure

The *enTish* infrastructure describes and composes services based on software agent technology. It was proposed by a research team at the University of Podlasie, in Poland [38]. The enTish infrastructure contains two specifications [38]: a language for describing services in an open and distributed environment, called *Entish,* and a protocol for performing the service composition, called *entish 1.0.*

EnTish considers a service as providing only one input/output operation, contrary to WSDL, which considers a service to be a collection of operations [40]. In enTish, the user's request, which is considered a task, is expressed by using the language Entish [38]. The user's request is sent to the middleware that is in charge of receiving requests, processing them, and returning the result to the user. This middleware is transparent to the user [39].

The enTish infrastructure is shown in Figure 3-3. It consists of the following elements [39]: (a) the Task Manager (TM): the user interface where users make requests, insert values into the required parameters and receive the resulting information; (b) the Service API: the element used to integrate external applications to treat them as services within the infrastructure; (c) the Entish Dictionary: the element used to integrate new data types, relations, and functions in the architecture; and (d) enTish middleware: the element that receives user requests, executes them, and returns the corresponding result.

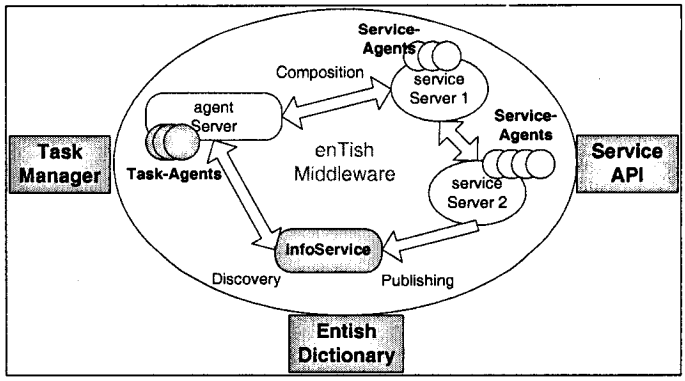


Figure 3-3. The enTish infrastructure [39]

The following sections describe enTish middleware, the Entish language, the entish protocol, and how enTish deals with the main issues related to Web service composition.

3.3.2.1 EnTish middleware

EnTish middleware is composed of software agents. The agents communicate with each other by exchanging messages whose content is expressed using the Entish language and following the entish 1.0 communication protocol. The agents contained in enTish middleware are *agent Server*, *task Agents*, *service Servers*, *service Agents* and *infoService*. [39]

The agent Server receives (from the Task Manager) tasks to be carried out and creates a corresponding task Agent for every task received. A task Agent is in charge of the task execution; it completes its functions as soon as the task is performed or when it detects that the task cannot be achieved. It has its own goal, intentions, and knowledge [40].

The service Servers integrate the services existing in the middleware. A service Agent is created for each service registered in a service Server that provides a specific operation. A service Agent has its own goal, commitments, and knowledge.

The infoService is used as a repository to store service information such as name, endpoint and type of operation offered by each service Agent.

3.3.2.2 Entish Language

The Entish language is used in enTish middleware for the following purposes [40]:

1. To describe the internal data (resources). It specifies the type, attributes, and location of data, as well as the methods used to manipulate them.
2. To express tasks. Tasks are specified as Entish formulas expressing abstract plans and initial conditions. A task can be either a user request or an operation offered by a certain service.
3. To describe service Agents and task Agents. Both service Agents and task Agents are characterized by their own goals, intentions and commitments, elements that represent their state.
4. To describe workflow and execution processes. Entish expresses the workflow required to carry out a task in a declarative way, without using actions. It also defines the messages exchanged between service Agents and task Agents during the execution process. These messages contain a field, called *order,* that distinguishes them from the messages that can exist within the protocol.
5. To express the content of messages exchanged among agents. Entish is the content language used in the messages exchanged by agents.

3.3.2.3 Description of service and task Agents

Service Agents, task Agents and Resources are the main entities inside enTish middleware [40]. Resources are data representations exchanged between service Agents and task Agents during a task execution. Service and task Agents are software agents that act as service providers and task suppliers, respectively. Entish describes the state of these agents according to the elements in Table 3.1.

Table 3.1. Description of the state of Agents [40]

Element	Type of Agent	
	task Agent	**service Agent**
Owner	Address of the task Agent	Address of the service Agent
Goal	**FormIn**: Empty or preconditions **FormOut:** Task of the agent	Type of operation performed by the service **FormIn**: Precondition of service invocation **FormOut**: Postcondition or effect
List of Commitments	Not used	Set of service commitments **FormIn:** Precondition of the commitment **FormOut**: Effect the service has committed to perform
Intentions	List of agent intentions **Plan.** Agent's plan **Workflow.** Intentions already committed with services **Performed.** Intentions already performed	Not used
Knowledge	Collection of INFO elements (evaluated formulas)	Collection of INFO elements (evaluated formulas)

As shown in Table 3.1, the state of the service and task Agents is defined by the following elements:

1. **Owner.** This element represents the location of the owner of the state.
2. **Goal.** This element consists of two Entish formulas: FormIn and FormOut. If the state belongs to a task Agent, the goal is the task that the agent performs. The FormIn formula can either be empty or contain an Entish formula expressing the preconditions for performing the task, and the FormOut formula contains the agent's task. If the state belongs to a service Agent, the goal represents the type of operation the service provides. The FormIn formula represents the preconditions for invoking the service, and the FormOut formula describes the postconditions or effects of service invocation.
3. **List of commitments.** This element is used by service Agents and contains elements called *commitments*. A commitment is represented by two Entish formulas: a FormIn formula, which contains the required preconditions of the commitment, and a FormOut formula, which contains the postconditions or "effects the service has committed to perform" [40]. As soon as the service Agent performs the commitment, it is removed

from the list of commitments and the information about its execution is incorporated into the knowledge element.

4. **Intentions**. This element is mainly used by task Agents and has three components: a *plan* element, which contains a list of intentions represented as Entish formulas that describe the agent's plan; a *workflow* element, which contains a list of intentions, moved from the plan element, that are already committed with services; and a *performed* element, which contains a list of already performed intentions.
5. **Knowledge**. This is a collection of *INFO* elements that represents the facts the agent knows, and is changed with every action the agent performs.

In enTish, a service is described according to the postcondition (effect) it produces based on specific preconditions.

3.3.2.4 The entish 1.0 protocol

The entish 1.0 protocol defines the type of messages that are exchanged between task Agents and service Agents, as well as the order in which they are sent when composing services [40]. Figure 3-4 shows how the entish 1.0 protocol is used when a task is performed by two collaborating services. A number that also defines the order of the message within the protocol represents the message type. The process for executing a user request is as follows:

1. A user requests the execution of task φ, represented as an Entish formula, from the Task Manager (TM).
2. The TM creates *taskAgent0* and assigns to it the execution of task φ.
3. TaskAgent0 sets task φ as its goal, by putting φ into its goal FormOut formula, and as its intention, by putting φ into the plan list.

4. TaskAgent0 sends a message to *infoService* to find a service that can perform task φ. In turn, infoService finds that *serviceAgent1* can provide φ and returns a message to taskAgent0 with the name of the service Agent.
5. TaskAgent0 sends a message to serviceAgent1 to request the execution of task φ. ServiceAgent1 realizes that, in order to perform φ, some preconditions must be fulfilled (the execution of task ψ). Then serviceAgent1 returns a message expressing its commitment to perform φ if task ψ is previously completed. Also, serviceAgent1 puts task φ on its list of commitments.
6. TaskAgent0 moves the intention φ from its plan list to its workflow list because serviceAgent1 has committed to execute φ. Then task ψ is put into the plan list representing the actual intention.
7. TaskAgent0 sends a message to infoService to find a service that can perform task ψ. InfoService finds that *serviceAgent2* can perform ψ and returns a message to taskAgent0 with the name of the service Agent.
8. TaskAgent0 sends a message to serviceAgent2 to request the execution of its intention ψ. ServiceAgent2 can unconditionally perform ψ; therefore, it returns a message expressing its commitment to execute ψ and puts ψ in its list of commitments.
9. TaskAgent0 moves the intention ψ from its plan list to its workflow list because serviceAgent2 has committed to execute ψ. The plan list is empty because there are no preconditions for performing ψ. The workflow list contains the intentions that the service Agents have already committed to and that represent the service composition to perform task φ. Then taskAgent0 simultaneously sends messages to the service Agents telling them that the workflow has been created and is ready to be executed.

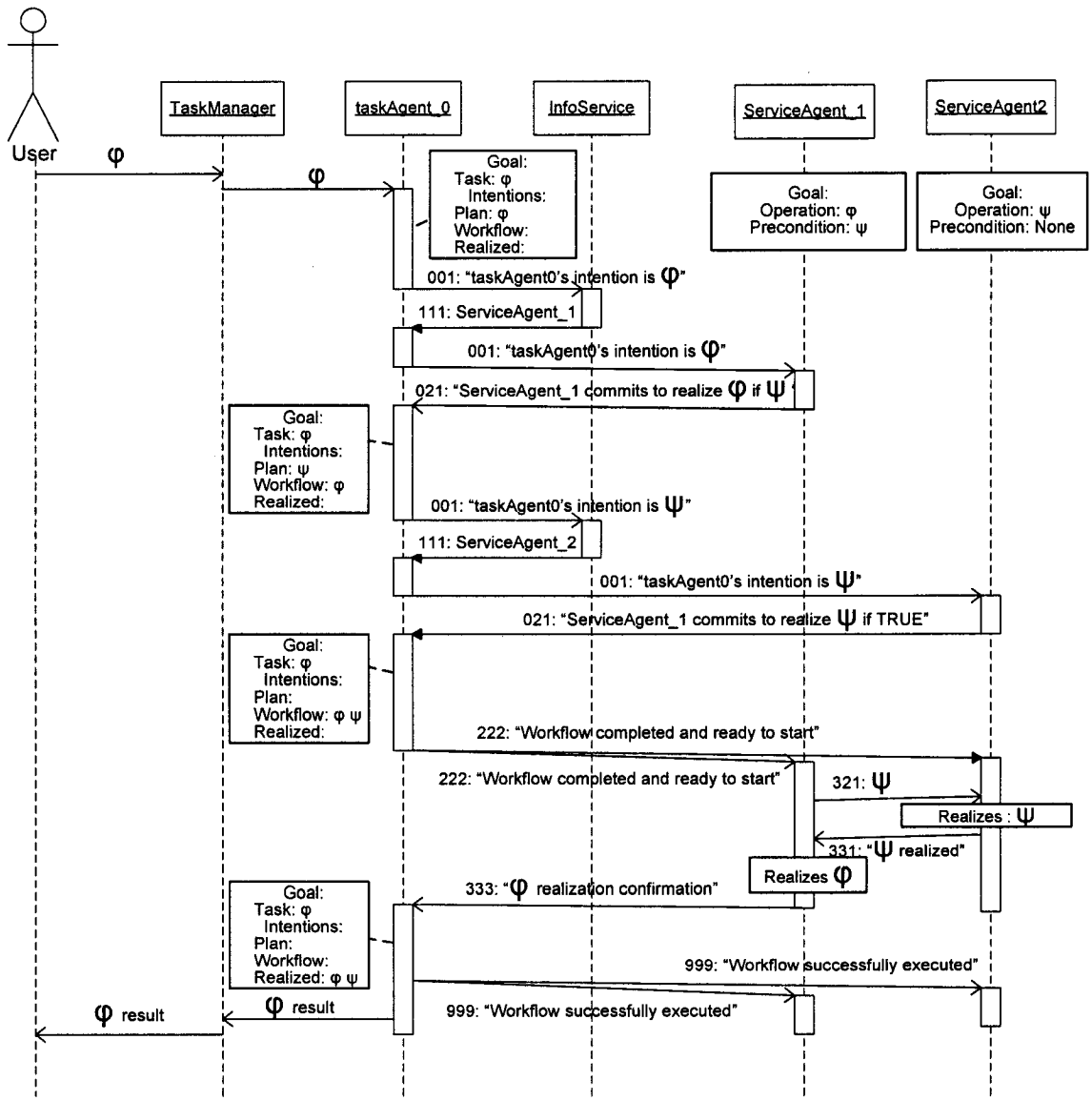


Figure 3-4. The entish 1.0 protocol

10. ServiceAgent1 executes the workflow by sending a message to serviceAgent2 to perform ψ. After completing ψ, serviceAgent2 returns a confirmation message. Then serviceAgent1 performs φ and sends a message to taskAgent0 confirming the successful execution of φ.

11. TaskAgent0 moves the intentions stored in its workflow list to its *performed* list. Then, it sends a final confirmation message to every service involved in the workflow so that

they can make the modifications to the resources permanent. At the same time, it sends a message to the TM confirming the successful execution of task φ.

12. The TM sends the result of the φ task execution to the user.

3.3.2.5 EnTish and Web service composition issues

This section presents how enTish solves the aforementioned problems associated with Web service composition.

3.3.2.5.1 Service orchestration

EnTish implements the reactive service orchestration in which services are assembled at runtime. The process that enTish follows for service orchestration consists of two phases [40]:

1. The user creates a request that is sent to the enTish middleware. The request uses the Entish language and contains the initial conditions and objective to achieve. The middleware creates generic plans to achieve the goal based on the initial conditions.
2. Within the middleware, one generic plan is selected. The services that can execute the plan are discovered and arranged into a workflow. The workflow and some conditions for executing the service are presented to the user. The user enters the values to fulfill the conditions and starts the workflow execution. The middleware executes the workflow, controls its execution and sends the response to the user.

3.3.2.5.2 Service choreography

With enTish, service choreography is implemented through the cooperation of service Agents without a centralized entity controlling the workflow execution. A set of service Agents commits to a workflow and then executes it.

3.3.2.5.3 Service discovery

With enTish, service discovery is performed in a centralized manner. The InfoService acts as a service repository where task Agents locate services that provide a specific operation.

3.3.2.5.4 Level of scalability and flexibility

The enTish infrastructure provides a low level of scalability. Only services that are integrated into the infrastructure through the Service API, and that are registered with the infoService, can be executed in a composition.

The enTish approach offers a medium level of flexibility. It defines services according to the effects they produce and the specific preconditions they require. The service to be incorporated into the composition is selected from among the services that provide the same effect (i.e., that have the same effect formula) and that are registered inside the infrastructure. Services outside the infrastructure cannot be considered for the composition.

3.3.2.5.5 Service communication

The communication mechanism among agents in enTish uses request-response messages. During the creation of the workflow, task Agents request a service Agent to commit to a task and wait for a response. A service Agent must also wait for the execution of another task that a different service Agent committed to during the workflow execution. [40]

3.3.2.5.6 Recursiveness

Recursiveness could be achieved if service Agents could act as task Agents. This way, a service Agent could ask the infoService about service Agents that provide a specific service, and could create a workflow that groups service Agents to perform the requested type of operation. The current version of enTish is not recursive because it does not use a list of commitments in the state of service Agents [40].

3.3.2.5.7 Definition of Interfaces

EnTish does not require a definition of interfaces. Services are defined by task formulas that describe the type of operations they perform.

3.3.2.5.8 Participation of atomic services

EnTish is a mandatory-composite service approach. Each service engaged in a workflow must be executed to successfully perform the service request.

3.3.2.5.9 Fault-recovery support

A fault-control mechanism is incorporated in enTish. If a service Agent participating in the workflow fails to complete a task, the service Agent sends a message to the task Agent, which then sends cancellation messages to all the participating service Agents. The cancellation triggers rollback operations [40].

3.3.2.5.10 Data representation

Data and control-flow information are handled together in enTish. Entish formulas contain preconditions and postconditions that express the values of data required for the execution of tasks. The data resulting from the task execution is stored in formulas; these formulas are used as the content of messages exchanged by service Agents during workflow execution.

3.3.3 DARPA Agent Markup Language for Services (DAML-S)

The DARPA Agent Markup Language for Services (DAML-S) [41] is a language and an ontology that is used to describe Web services with sufficient information to enhance the processes of "discovery, invocation, composition, and execution monitoring." With DAML-S, Web services are "computer-interpretable." DAML-S enhances the current Web Services Architecture by incorporating the ideas of the Semantic Web.

3.3.3.1 The Semantic Web

The Semantic Web is an effort to create Web pages with information structured in such a way that the Web content is easily manipulated and interpreted not only by humans, but also by software applications.

The implementation of the Semantic Web is based on the following elements [42]:

1. Languages to semantically express the information contained on Web pages and make it easy to interpret by software applications.
2. Knowledge representations to provide software applications with the ability to deduce facts from a context, based on information about existing objects.
3. Ontologies to define the relationship between concepts in a specific context. Inference engines use ontologies to detect the existent relationships between objects to provide information to trigger actions.

Some languages used to create the Semantic Web are the Resource Description Framework (RDF), a W3C standard; DAML-OIL (Ontology Inference Layer) that extends RDF; and the OWL Web Ontology Language. Some tools for creating Semantic Web applications include JENA (an HP project) which provides support for RDF, DAML-OIL, and OWL [43].

By applying the Semantic-Web based ideas to the Web services world, "Autonomous Semantic Web Services" are created, which enhance the "discovery, invocation, composition and execution monitoring processes" [44].

3.3.3.2 DAML-S

DAML-S extends the DAML-OIL ontology to enhance the description of Web services as well as the relationships among them [45]. DAML-S fits in the Web service stack framework as shown in Figure 2-9. DAML-S consists of three components [45]: a *service model*, a *service*

profile, and *service grounding*. DAML-S uses WSDL to describe Web services and incorporates semantics into the Web service description by using RDF.

The service profile describes the service by adding semantic information to enhance the service discovery. It contains the service name, a textual description, information about the provider, a rate of quality, and the description of the offered functionality, including the input and output parameters as well as their preconditions and effects.

The service model contains information about the process governing the execution of a Web service. It contains the sequence in which the operations of involved atomic Web services are invoked. The information in the service model component enables the "automatic composition and execution" of the Web service [41].

Finally, service grounding associates the exchanged messages defined in the process model with their corresponding representation in WSDL or SOAP.

3.3.3.3 DAML-S and Web services composition issues

This section describes how DAML-S solves the aforementioned problems associated with Web service composition.

3.3.3.3.1 Service orchestration

DAML-S implements reactive service orchestration. Atomic Web services that participate in a composite service are dynamically discovered and invoked at runtime [44].

DAML-S assumes that every available Web service has its own DAML-S description, which includes the definition of the input and output parameters as well as the preconditions and effects (postconditions), collectively known as the IOPEs, to execute the Web service. Instead of describing the specific services to interact with (as BPEL4WS does), a composite DAML-S Web service contains the description that best identifies its atomic DAML-S Web services. At

runtime, the composite DAML-S Web service discovers atomic DAML-S Web services that best fit the description (profile) of the desired service, and then dynamically invokes it. The orchestration of services in the DAML-S approach is represented as a workflow described with the service model component of DAML-S [44].

DAML-S defines a composite service by defining services as one of three kinds of processes [45]: an *atomic*, *composite*, or *simple process*.

DAML-S atomic processes are atomic services. They are able to execute simple operations, such as remote-procedure calls. Each one is defined by the operation it performs as well as its inputs, outputs, preconditions, and effects.

A DAML-S composite process integrates atomic processes into a workflow. It is defined by using the following control constructs [46]:

- **Sequence**: Specifies a list of atomic or composite processes that are sequentially executed.
- **Split**: Specifies a set of processes concurrently executed without a final synchronization point.
- **Split-Join**: Specifies a set of processes concurrently executed and synchronized after execution at a final synchronization point.
- **Unordered**: Specifies a set of processes executed without a specific order. All the processes must be executed to successfully carry out this construct.
- **Choice**: Executes a subset of processes selected from a group by using certain selection criteria.
- **If-Then-Else**: Executes the process corresponding either to the *then* clause or to the *else* clause based on the evaluation of a *condition*.

- **Iterate**: Provides repeated execution of one or more processes. The repetition is controlled by either a *repeat-while* construct, which executes a set of processes *while* a certain condition is valid, or by a *repeat-until* construct, which executes a set of processes *until* a certain condition becomes true.

Finally, a simple process encapsulates atomic and composite processes into a definition to hide the involved operation details and provide a black-box view of the entire process.

3.3.3.3.2 Service choreography

The service choreography is implemented in a centralized way when using DAML-S. The architecture to create DAML-S-enabled Web services consists of three interrelated modules [44]: the Web service invocation, the DAML parser, and the DAML-S Virtual Machine (VM). The Web service invocation module is in charge of invoking individual Web services. After receiving a response message, it sends the information that is extracted from the received message to the DAML parser. Then, the DAML parser produces predicates from the information received by using DAML-OIL ontologies and DAML-S descriptions of other services, which the DAML-S VM module can use to deduce facts and then act accordingly.

The DAML-S VM decides which flow of action to follow based on the workflow defined in the DAML-S process model part [44]; the semantics of the workflow are implemented as a set of rules to be manipulated by the DAML-S VM.

3.3.3.3.3 Service discovery

Atomic services are discovered and integrated into a workflow at runtime. The entity that is in charge of advertising and receiving search requests is the DAML-S/UDDI Matchmaker [44]. The DAML-S/UDDI Matchmaker is an architecture that combines UDDI for Web service advertisement purposes and DAML-S for expressing Web services and for enhancing the matching process when seeking a Web service with a specific functionality.

The service discovery process is summarized in the following steps [44]: (1) Web service providers create DAML-S-enabled Web services by using DAML-S for service description. The service description includes the input and output parameters as well as the preconditions and its effects; (2) DAML-S-enabled Web services are published in the DAML-S/UDDI Matchmaker. The Web service definitions are stored as service profiles; (3) A Web service seeking another service that provides a specific functionality creates a profile of the "ideal service it wants to interact with." It sends a request to the DAML-S/UDDI Matchmaker with the service profile; (4) The DAML-S/UDDI Matchmaker searches for service profiles that semantically match the requested profile; (5) The DAML-S/UDDI Matchmaker returns a message with the information about the matched Web service to the requester, which can invoke the Web service.

3.3.3.3.4 Level of scalability and flexibility

DAML-S provides a medium level of scalability. Only DAML-S-enabled Web services can participate in the composition. A service provider must describe its services by using DAML-S and register them in the DAML-S/UDDI Matchmaker to be incorporated in the composition.

The level of flexibility with DAML-S is high. A DAML-S-enabled Web service can be selected from a set of services that provide the same functionality by taking into account parameters such as the quality of service.

3.3.3.3.5 Service communication

DAML-S-enabled Web services can be contacted by using both request-response and one-way messages. Each received message is delivered from the Web service invocation module to the DAML parser module. The DAML-S VM uses inference rules to deduce the kind of message received and act according to the workflow described in the service model.

3.3.3.3.6 Recursiveness

A DAML-S-based Web service can act as an atomic service and take part in a broader composite service.

3.3.3.3.7 Definition of Interfaces

The definition of a DAML-S-enabled Web service not only includes the operations that the service offers but also the preconditions for and effects of invoking the service. A service profile contains the definition of a DAML-S-enabled Web service in a semantic form. The process of discovering a Web service uses its service profiles to find, from a set of services, the one that semantically matches the desired profile the best. The Web services with semantically matching profiles do not need to have the same interface. Therefore, the definition of Web service interfaces is not applicable to DAML-S.

3.3.3.3.8 Participation of atomic services

DAML-S is a mandatory-composite service approach. The execution of a composite service depends on the successful execution of all the services engaged in the workflow.

3.3.3.3.9 Fault-recovery support

The current version of DAML-S does not provide support to deal with failures in the invocation of Web services involved in the composition [41].

3.3.3.3.10 Data representation

The DAML-S service model includes the data representation and the control flow together.

3.3.4 CompoSing WEb accessibLe inFormation and buSiness sERVices (SELF-SERV)

SELF-SERV is an infrastructure for Web service composition that dynamically integrates Web services into a workflow and coordinates them in a peer-to-peer manner [47]. It aims to create

a scalable infrastructure for Web service composition where a large number of services can be integrated into the infrastructure and take part in the composition process.

SELF-SERV distinguishes three kinds of services: *elementary services*, *composite services*, and *service communities* [47]. Elementary services are equivalent to atomic services in that they execute individual operations. A composite service is a collection of atomic services that offers a functionality that represents the integration of the individual functionalities of its components. Service communities congregate elementary or composite services that offer the same functionality (i.e., the same service operation). In order to be part of the composition process, elementary and composite services must be registered with the service community that represents the operation they provide.

Every service (elementary, composite or community) has an interface that is implemented by its *service wrapper*. A service wrapper is a software entity deployed by the service provider that receives invocation requests [48]. It can be an elementary, a composite or a community wrapper based on the service it implements. The invocation of any service is through its wrapper.

3.3.4.1 SELF-SERV and Web service composition issues

This subsection presents how the SELF-SERV infrastructure solves the previously discussed problems associated with Web service composition.

3.3.4.1.1 Service orchestration

SELF-SERV implements the reactive service orchestration approach [47]; services are selected and integrated into the composition only when they are needed. SELF-SERV models the behaviour of a composite service by using a *statechart*. A statechart contains the operations that must be invoked by elementary services, without referring to specific service providers, as well as the order in which they are to be invoked.

Figure 3-5 shows how SELF-SERV models a composite travel service by using statecharts. A statechart models the behaviour of a system by representing the different states a system can achieve as well as the events that trigger the transition between states. A statechart has an initial state and one or more final states.

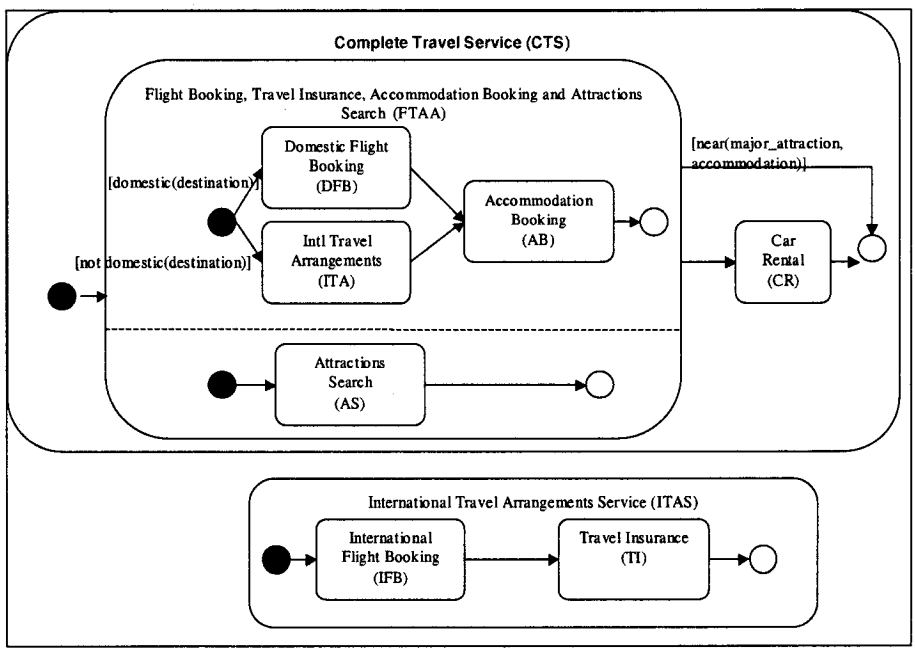


Figure 3-5. SELF-SERV travel service [47]

SELF-SERV defines two kinds of states: *basic* and *compound*. A basic state represents a specific service, either atomic or composite. The label of the basic state represents the service operation that is invoked once the state is reached. When a basic state is reached, the service community that is in charge of providing the service operation receives a request to execute the operation; then, it selects one of its members, by following certain selection policies, and delegates the service invocation to it. The statechart shown in Figure 3-5 contains six basic states: Domestic Flight Booking (DFB), Accommodation Booking (AB), Attractions Search (AS), Car Rental (CR), International Flight Booking (IFB), and Travel Insurance (TI).

Compound states enclose one or more statecharts. If a compound state contains only one statechart, it is called an *OR-state*; otherwise, it is called an *AND-state* and represents the concurrent execution of its enclosed statecharts. The execution of a compound state is achieved when all of its final states are reached [47]. The statechart in Figure 3-5 contains two compound OR-states, CTS and ITAS, and one compound AND-state, FTAA. The events triggering the transition between states represent the conditional execution of services. For instance, on the statechart in Figure 3-5, the Domestic Flight Booking operation is only invoked if the evaluation of the Boolean function, *domestic(destination)*, is true. In addition, the Car Rental operation is only invoked if the *near(major_attraction, accommodation)* function is false. Thus, SELF-SERV only provides support for the sequential, concurrent and conditional control-flow operations.

3.3.4.1.2 Service choreography

Service choreography in SELF-SERV is based on the cooperation of distributed *state coordinators*. A state coordinator is a software entity that represents one state in the statechart, and that collaborates with other state coordinators in a peer-to-peer fashion to execute the composite service modeled by the statechart. State coordinators are deployed by each of the participating service providers [48]. A state coordinator has the following responsibilities: (1) starting the execution of its service at the moment that all its preconditions are met; (2) sending a completion notification to state coordinators that are likely to be the next ones executed; (3) receiving notifications of external events to decide whether the service execution must be cancelled based on the received events, and then interrupting the execution of uncompleted actions; and (4) notifying the state coordinator that is likely to be the next one executed about the cancellation of the execution.

The invocation of any service, either elementary, composite, or coordinator, is performed by its corresponding wrapper. The invocation of a wrapper depends on the kind of service it implements. An elementary wrapper invocation executes the operation that the elementary service implements. A composite wrapper starts its invocation by sending a control-flow notification message to all the initial state coordinators of each statechart. In turn, the state coordinators collaborate with each other until the final state is reached. Then the composite wrapper receives a control-flow notification message indicating the execution of all its operations. A community wrapper is invoked by selecting, from the services inside the community, the one that best satisfies the selection policies. Then the selected service is invoked through its wrapper.

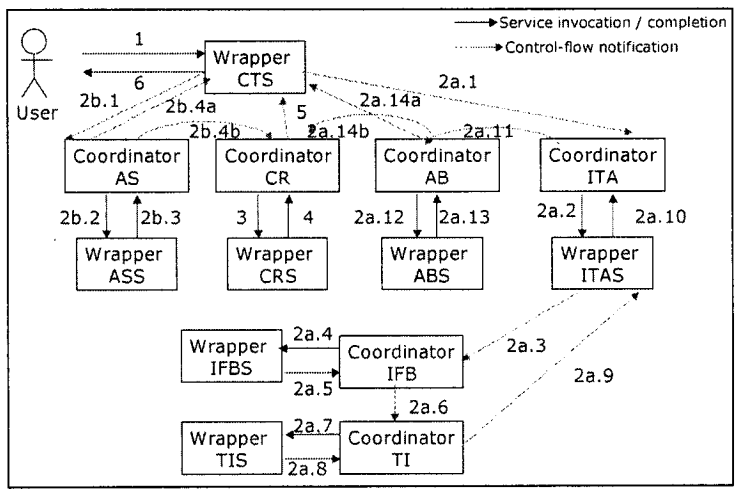


Figure 3-6. SELF-SERV services choreography [47]

Figure 3-6 shows the interactions among state coordinators involved in the execution of the travel service described in the statechart of Figure 3-5. In addition, Figure 3-6 depicts the two kinds of messages exchanged among participating software entities: *control-flow notification* messages and *invocation-completion* messages.

The service execution starts when the composite wrapper of the service CTS (the top-level state) receives a service invocation message from the user. The CTS wrapper sends a

control-flow message to all coordinators of the initial states in the CTS state (DBF, ITA, and AS). In the case of an international flight, the ITA and AS states are activated (the coordinator of the DBF state is not shown in Figure 3-6). After fulfilling the preconditions, the coordinators of the ITA and AS states send an invocation message to the ITA and AS wrappers, respectively. Because AS is an elementary service, the AS wrapper directly invokes the Attractions Search operation and returns a completion message to the AS coordinator, who then sends a notification message to the CTS wrapper, to inform it of the AS service execution, and a control-flow message to the next possible state to be reached (CR). Meanwhile, the composite ITA wrapper sends a control-flow message to the coordinator of the first state in its statechart (IFB). The IFB coordinator invokes the IFB wrapper and receives a notification message, which triggers a control-flow message to the coordinator of the following state (TI). After receiving the message, the TI coordinator invokes the TI wrapper and receives a completion message. Then, the TI coordinator realizes that the final state of the statechart (ITA) has been reached and returns a notification message to the ITA wrapper, which, in turn, sends a completion message to its coordinator. The ITA coordinator realizes that the following state to reach is AB, and it sends a control-flow message to the AB coordinator, which then invokes the AB wrapper. After receiving a completion message from the AB wrapper, the AB coordinator realizes that the final state of the statechart has been reached and sends a notification message to the CTS wrapper and a control-flow message to the coordinator of the next possible state to be reached (CR). The CR coordinator has all the information to fulfill its preconditions and then invokes the CR wrapper. After receiving a notification message from its wrapper, the CR coordinator realizes that the final state has been reached and returns a notification message to the CTS wrapper. Finally, the CTS returns a completion message to the user with the result of the composite service invocation.

3.3.4.1.3 Service discovery

The SELF-SERV infrastructure contains a module that acts as a service *discovery engine* [47] and that uses a service repository called a *service catalogue*. The service catalogue stores the service descriptions, which contain their location and properties. The discovery engine is queried by service wrappers to locate services that provide the operations of the state they represent. If several services provide the same operation, a selection policy is used to determine the best service to be incorporated in the composition. It considers certain parameters in the selection process, such as the cost and the response time [49].

3.3.4.1.4 Level of scalability and flexibility

SELF-SERV provides a high level of scalability [47]. A service community can congregate a large number of services offering the same functionality (i.e., the same operations). Web services can join or leave the infrastructure by contacting the service community. Matching between the operations of the services and the operations of the service community is performed at registration time in order to increase the flexibility of the infrastructure. However, SELF-SERV can only integrate service providers that host a state coordinator and a service wrapper for every Web service operation to be deployed. Thus, service providers must make the extra effort to create state coordinators and service wrappers in order to incorporate their services into the composition process.

SELF-SERV offers a high level of flexibility. A service community selects, from a set of registered services, the one that best fits the service composition according to certain selection policies.

3.3.4.1.5 Service communication

In SELF-SERV, the service invocation can be performed by using request-response messages (remote method invocation) or one-way messages (message exchange) [48]. A wrapper contains information about the protocol used in the invocation of operations.

3.3.4.1.6 Recursiveness

SELF-SERV can recursively define composite services. With SELF-SERV, a composite service can contain elementary and composite services and "a service community can be registered with another community" [48].

3.3.4.1.7 Definition of Interfaces

The definition of service interfaces in SELF-SERV is important because services are registered in service communities that integrate services offering the same operations [47]; this is also the basis for greater scalability and flexibility. By using selection policies, a community service chooses, from a set of services that provide the same interface, the one that best fits in the composition.

3.3.4.1.8 Participation of atomic services

SELF-SERV does not support negligible atomic services within a process. A SELF-SERV composite service is a mandatory-composite service. The successful execution of it depends on the successful execution of each one of its atomic services.

3.3.4.1.9 Fault-recovery support

SELF-SERV provides fault-recovery support by describing the transactional perspective of a composite service, which guarantees the correct execution of all its component services [48]. This includes transaction-monitoring techniques for transactional support.

3.3.4.1.10 Data representation

SELF-SERV allocates the data-flow together with the control-flow. Intermediate data is exchanged among services by using the same control-flow message. Services face a data conversion process to manipulate the information received [48].

3.4 Comparison of Web service composition techniques

The main issues in developing composite Web services were pointed out in Section 3.2. These issues were used in Section 3.3 to compare the most relevant Web service composition techniques. Table 3.2 shows how the described service composition approaches address these issues.

Table 3.2. Comparison of Web service composition approaches

Parameter	Web service composition approach			
	BPEL4WS and related technologies	**enTish**	**DAML-S**	**SELF-SERV**
Service orchestration	Proactive	Reactive	Reactive	Reactive
Service choreography	Centralized	Cooperation of distributed entities	Centralized	Cooperation of distributed entities
Service discovery	At development time by using UDDI registries	Dynamic at runtime by using InfoService.	Dynamic at runtime by using the DAML-S/ UDDI-based service repository	Dynamic at runtime by using the service catalogue
Level of scalability	Non-existent	Low	Medium	High
Level of flexibility	Non-existent	Medium	High	High
Service communication	Request-response and one-way messages	Request-response messages	Request-response and one-way messages	Request-response and one-way messages
Recursiveness	Supported	Not supported	Supported	Supported
Definition of interfaces	Not applicable	Not applicable	Not applicable	Important
Participation of atomic services	Mandatory-composite service	Mandatory-composite service	Mandatory-composite service	Mandatory-composite service
Fault-recovery support	Provided	Provided	Not provided	Provided
Data representation	Data and control flow together	Data and control flow together	Data and control flow together	Data and control flow together

An analysis of Table 3.2 reveals the following significant points:

1. Most service composition techniques implement the reactive service orchestration, except BPEL4WS and its related technologies, whose composition statically describes the final integrated services.
2. Only enTish and SELF-SERV incorporate software entities that cooperate to execute a composite service. BPEL4WS and DAML-S require a centralized entity to coordinate the involved services.
3. All the composition techniques incorporate a centralized service repository for operating the publishing and searching services. Only BPEL4WS does not provide a dynamic service-discovery process at runtime.
4. While BPEL4WS presents the lowest level of scalability and flexibility, SELF-SERV presents the highest levels of these two parameters. However, SELF-SERV requires that service providers host both a state coordinator and a service wrapper for each Web service operation they provide. Thus, service providers must make an extra effort to incorporate their services into the infrastructure, because the service discovery process only locates services that are registered in the infrastructure.
5. Most of the composition techniques support communication among services by using request-response and one-way messages, except enTish, which only communicates with request-response messages.
6. EnTish does not support a recursive definition of composite services.
7. Only SELF-SERV emphasizes the definition of the interfaces of the participant services. The service-selection process is based on the fact that there are sets of services offering the same functionality through similar interfaces but with different usage restrictions.

8. All the techniques are mandatory-composite services; they do not allow atomic services to fail. If an atomic service fails, the execution of the composite service fails.
9. The current version of DAML-S does not have fault-recovery support to handle faulty composite services.
10. All the techniques handle data and control-flow together.
11. Only BPEL4WS and DAML-S use UDDI service registry to look for available services. SELF-SERV and Entish use private service repositories for service discovery.

3.5 Summary

Web services allow companies to offer their software functionalities to external companies by using standard and open technologies. Web Services Architecture is facing different issues in its effort to become widely accepted and, therefore, widely used. One of the crucial issues facing Web service technology is Web service composition, which focuses on solutions to integrate individual Web services into the creation of customized applications. With Web service composition, companies can create partnerships to solve complex problems by integrating individual operations.

This chapter covered the definition of, and the primary issues facing, Web service composition. Next, the most relevant service composition approaches were defined and analyzed according to those issues. Finally, a comparative analysis of composition techniques was presented.

Chapter 4
Design of an Agent-Based Web Service Composition Framework

Current Web service composition techniques present a set of solutions to the service composition challenge. Companies can select the solution that best fits their requirements. A solution similar to that offered by the family of BPEL4WS is adequate for situations where partnerships are static and services are always provided for the same partner. Alternately, if a company is looking for a solution that dynamically selects the most suitable service provider, then a dynamic approach, such as SELF-SERV, is necessary. In addition, final users have to analyze the levels of scalability and flexibility required for selecting a particular service composition approach.

A Web service composition technique is scalable if it allows more services to be integrated into an already existing collection of services. In this way, several services can offer the same operations and be deployed by a varying number of providers. The integration of new services is intended to require minimal effort from the provider.

In addition to its level of scalability, a Web service composition approach is characterized by its level of flexibility. Flexibility is related to the capability of the composition process to select, from among atomic services that offer the same functionality, the one that best provides the functionality according to the current requirements.

Although current dynamic Web service composition techniques include acceptable levels of scalability and flexibility, the effort that a service provider has to put forth is considerable; the provider not only has to create the software program that carries out the specific functionality, but also has to describe its services and register them by using non-standard

languages and repositories (as in the DAML-S solution) or it has to host a module of the service composition architecture in order to be integrated into the architecture (as is required by SELF-SERV).

This chapter presents the design of an Agent-based Web Service composiTion framEworK (ASTEK) that maintains the high levels of scalability and flexibility offered by other approaches and avoids the extra work a service provider must perform to be integrated into the service composition process.

The remainder of this chapter is organized as follows. Section 4.1 introduces ASTEK. Section 4.2 presents the travel service scenario as an example of a software application that benefits from ASTEK. The use of this specific scenario does not cause a loss of generality; ASTEK can be used in a variety of situations. Section 4.3 describes the language that defines the composite services that can be executed in ASTEK. Section 4.4 describes the representation of the data that is produced by the atomic-service invocations and used as intermediate data for executing subsequent services. Section 4.5 details the protocol that is used by software agents during composite service execution. Section 4.6 presents how ASTEK solves problems related to Web service composition. Finally, Section 4.7 summarizes the concepts presented in this chapter.

4.1 Introduction

The Agent-based Web Service composiTion framEworK (ASTEK) is an infrastructure for Web service composition that is based on software agent technology. It associates a service description with a business process specification by using a business process language. It coordinates services in a peer-to-peer fashion where a set of software agents is responsible for executing the process. In addition, ASTEK provides high levels of scalability and flexibility in the incorporation of atomic Web services into the infrastructure.

ASTEK not only considers the incorporation of Web service operations into the service composition, but also the incorporation of the functionality offered by software agents representing legacy applications.

ASTEK is a reactive, agent-based Web service composition approach that has the following advantages over the previously analyzed service composition solutions:

1. The Web service discovery process is dynamically performed at runtime by using the standard UDDI repository.
2. The composition process allows for the failure of some services without affecting the entire execution by incorporating optional-composite services.
3. The basic Web service standards (i.e., WSDL, UDDI, and SOAP) are widely used in the composition approach. ASTEK is an alternative that uses minimal semantics.
4. Service providers can easily incorporate their services into the composition process with minimal effort. As a result, the level of scalability is increased.
5. Software entities involved in the composition process select the service to be invoked by using the user information and the current intermediate data. Consequently, the level of flexibility is increased.

ASTEK is designed with the following features:

1. Communication among Web services is carried out by using remote procedure calls.
2. A single Web service provides three operations: one that represents its functionality, one that cancels the execution of its functionality, and a *ping* operation that monitors the availability of the service.
3. Data and control flow representations are handled separately in two different artifacts. However, both are exchanged among software entities by using the same message.

4. Users that want to invoke services must already be registered in the system.

ASTEK shares some similarities with the previously analyzed service composition methods. First, it uses a business process representation that is similar to BPEL4WS. Second, it uses a coordination protocol that organizes the involved software agents, as in enTish. Finally, it dynamically discovers atomic services and incorporates them into the process execution at runtime, as is done by DAML-S and SELF-SERV.

4.2 Travel service scenario

In the travel service scenario a user makes a request to the system (ASTEK) to arrange a two-way trip to a certain destination during a certain period of time. The travel service is composed of the atomic services for *Flight Booking*, *Hotel Booking*, *Travel Insurance*, *Car Rental*, and *Attractions Search*. The atomic services are deployed as Web services and are registered in a UDDI service registry.

Figure 4-1 shows the use case diagram that models the travel service scenario. Each atomic service can be provided by one or more Web services. ASTEK considers that a Web service offers a single operation with its corresponding cancellation operation. As a result, a Flight Booking Web service executes the *flightBooking* operation and its corresponding cancellation operation. Web services also have a ping operation that ASTEK uses to monitor their availability.

The travel service is invoked with the following parameters: the *minimum* and *maximum departure* and *return dates*, the *origin*, the *destination*, and some information about the user. The travel service invocation returns the details of the trip to the user. Appendix C contains a detailed description of the atomic services involved in the travel service.

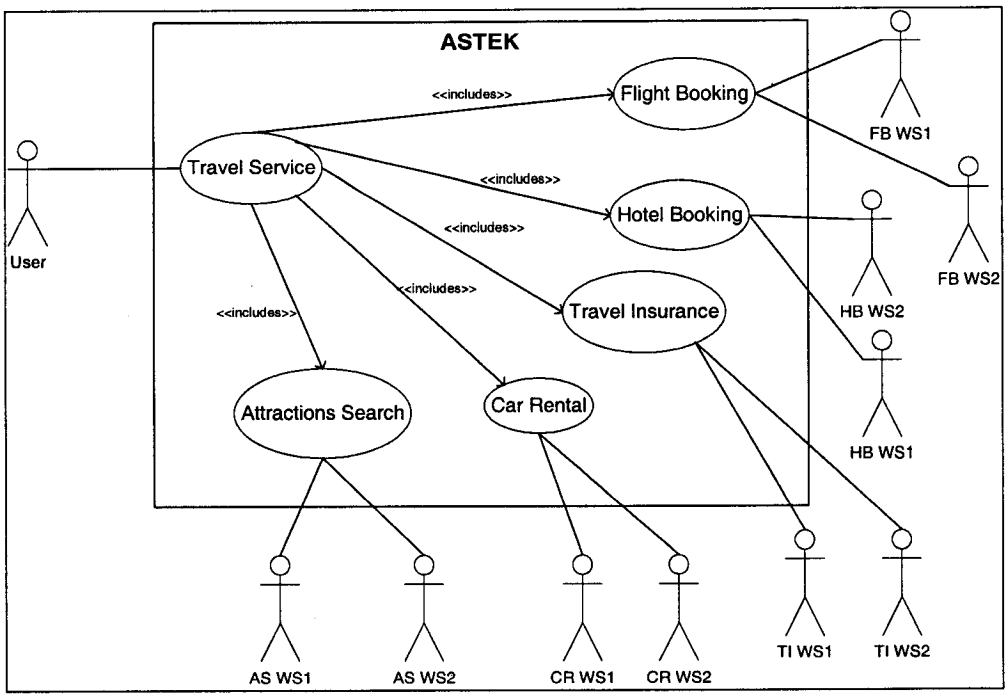


Figure 4-1. Travel service use case diagram

In ASTEK, the user must be registered in the system before being able to invoke an available service. The user information stored in the system is the user's name, occupation, date of birth, e-mail address, physical address (number, street, city, province, postal code and country), and personal preferences, such as his or her favourite mode of transport (car, plane, train, etc.), seating preferences (window, aisle), time of trip (morning, afternoon, evening), and types of attractions (movies, amusement parks, etc.).

4.3 Process modelling language

In ASTEK, a composite service is a business process that is modelled as an activity diagram. An activity diagram models the behaviour of a system by using a diagram that contains nodes, which represent an action, inner activity or decision point; and edges, which represents both control and data flow. Activity diagrams are widely used for representing business processes due to the explicit definition of the sequence, concurrency, and logic

control of the activities involved in the process [50]. For the purposes of this discussion, the terms *composite service*, *business process*, and *process* are related to the same concept and will be used interchangeably.

An action is the minimal executable element in an activity diagram, and is executed only if all its preconditions are fulfilled. Once an action is successfully executed, the execution of its successors is enabled; they use the resultant output as the inputs for their execution [50].

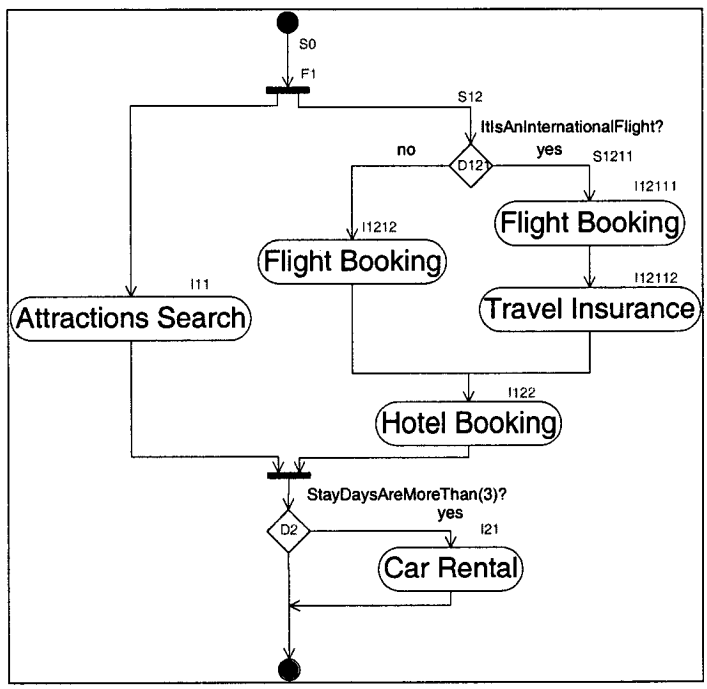


Figure 4-2. Travel service business process with identifiers for each activity

Figure 4-2 shows an activity diagram that models the integration of individual services into the travel service. The diagram contains nodes to indicate the concurrent execution of activities, decision points to regulate the control and data flow, and actions to represent the service invocations. It also shows the sequence of activities involved in the process.

The syntax of the process modelling language used in ASTEK is described by the production rules expressed in the regular tree grammar (RTG)[1] of Figure 4-3(a).

The definition of a business process starts with the *process* symbol, which produces a *sequence, flow, decision* or *invoke* activity. A sequence activity represents the consecutive execution of flow, decision, or invoke activities, whereas a flow activity denotes the concurrent execution of sequence, flow, decision or invoke activities. A decision activity indicates the execution of a *then* or an *else* activity based on the evaluation of the predicate—a terminal symbol—contained in its *if* part. A *then* or an *else* activity denotes the execution of one of the sequence, flow, decision, or invoke activities. The invoke activity, also a terminal symbol, denotes the service being invoked. The terminal symbols used in the travel service scenario are shown in Figure 4-3(b).

(a) Regular Tree Grammar for the process modelling language used in ASTEK

G_{ASTEK} = (N,T,S,P)
N = {process, sequence, flow, decision, invoke, service}
T = {predicate, parameter, service}
S = {process}
P = { process → sequence | flow | decision | invoke ,
sequence → (flow | decision | invoke) + ,
flow → (sequence | flow | decision | invoke) + ,
decision → if then (else |Ɛ) ,
if → predicate (parameter)* ,
then → (sequence | flow | decision | invoke) ,
else → (sequence | flow | decision | invoke) ,
invoke → service ,
predicate → PcData ,
parameter → PcData ,
service → PcData
}

(b) Terminal symbols for the travel service scenario

predicate → "ItIsAnInternationalFlight" | "StaysDaysAreMoreThan"
service → "FlightBooking" | "HotelBooking" | "TravelInsurance" | "CarRental" | "AttractionsSearch"

Figure 4-3. Grammar of the process modelling language used in ASTEK

[1] A Regular Tree Grammar (RTG) is used in computer science to describe the syntax of languages. It expresses both the symbols (strings) and the combinations of them that are allowed in the language. An RTG G=(N,T,S,P) consists of a finite set of non-terminal symbols (N), a finite set of terminal symbols (T), a non-terminal symbol that is used as the starting symbol (S), and a finite set of production rules (P) [51].

The grammar G_{ASTEK} is then expressed in a XML Schema as the language that defines business processes inside the ASTEK infrastructure. The resulting XML Schema is shown in Appendix D. It is used to create business processes in the form of XML documents. The XML Schema contains a definition of the elements allowed in a process XML document as well as the attributes of each element.

The following subsections describe the elements and attributes of the XML Schema shown in Appendix D.

4.3.1 Process element

The **<process>** element is the initial element in the definition of a business process. It is composed of one of the **<sequence>**, **<flow>**, **<decision>**, or **<invoke>** elements. Its attributes are the following:

1. **ID**: a required attribute that distinguishes this process from others
2. **name:** a required attribute that contains the name of the represented process
3. **status:** a required attribute that contains the status of the process execution. Before being initiated, the process has an *inactive* status. During execution of the process, its status is *active*. The process status is *done* if the process is successfully completed and *failed* in the case of a failure in its execution.
4. **startingTime***:* the date and time when the process starts its execution
5. **endingTime**: the date and time when the process is successfully completed
6. **processingTime***:* the number of milliseconds that the process took to be executed
7. **failureCause***:* an explanation of what caused the process execution to fail

4.3.2 Sequence element

The **<sequence>** element encompasses activities that are sequentially executed. It can contain one or more **<flow>**, **<decision>** or **<invoke>** elements. Its attributes are the following:

1. **level:** a required attribute that distinguishes the **<sequence>** activity from all other activities in the process
2. **status:** the status of the **<sequence>** execution. Before being executed, the status of the **<sequence>** is *inactive*. During the execution of the activity, it has the status of *processing*, and after a successful execution it has the status of *done*. If some of its inner elements cannot be executed, its status is *failed*.

4.3.3 Flow element

The **<flow>** element includes activities that are simultaneously executed. It can contain one or more **<sequence>**, **<flow>**, **<decision>** or **<invoke>** elements. Its attributes are the following:

1. **level:** a required attribute that distinguishes the **<flow>** activity from the all other activities in the process
2. **status:** the status of the flow execution. Before being executed, the **<flow>** activity is *inactive*. During execution, it has the status of *processing*, and after a successful execution it has the status of *done*. If some of its inner elements cannot be executed, its status is *failed*.
3. **lastOnFinish:** the level of the inner activity that finishes its execution last
4. **timeOnFinish:** the date and time when all the inner elements finished their execution

4.3.4 Decision element

The **<decision>** element encompasses the following elements: an **<if>** element, a **<then>** element and possibly an **<else>** element. The inner activities of the **<then>** element are executed if the predicate in the **<if>** element becomes true; otherwise, the contents of the **<else>** element (if it exists) are executed. The **<decision>** element's attributes are the following:

1. **level:** a required attribute that distinguishes the **<decision>** activity from all other activities in the process
2. **status:** the status of the **<decision>** execution. Before being executed, the **<decision>** activity is *inactive*. During execution, it has the status of *processing*, and after a successful execution it has the status of *done*. If validation of the predicate in the **<if>** element fails, or if some of the activities in the **<then>** or **<else>** elements cannot be executed, its status is *failed*.
3. **validation:** contains the result of the evaluation of the conditional clause included in the **<if>** element. The permitted values are: *undefined*, if the clause has not yet been evaluated; *true*, if the conditional clause is positive; and *false*, if it is negative.

4.3.4.1 If element

The **<if>** element contains a **<predicate>** element and zero or more **<parameter>** elements. The **<predicate>** is a terminal element with the conditional clause to be evaluated. In the case of the travel service, it can be either the *ItIsAnInternationalFlight* string or the *StayDaysAreMoreThan* string. The **<parameter>** element contains the information required to evaluate the predicate (e.g., the *StayDaysAreMoreThan* predicate requires the number of days of the stay).

4.3.4.2 Then element

The **<then>** element is composed of one of the **<sequence>**, **<flow>**, **<decision>** or **<invoke>** activities. Its contents are executed if the conditional clause in the **<predicate>** element is evaluated as *true*.

4.3.4.3 Else element

The **<else>** element is composed of one of the **<sequence>**, **<flow>**, **<decision>** or **<invoke>** activities. Its contents are executed if the conditional clause in the **<predicate>** element is evaluated as *false*.

4.3.5 Invoke element

The **<invoke>** element contains the **<service>** element that defines the action (a service operation) to be executed. It has the following attributes:

1. **level:** a required attribute that distinguishes the **<invoke>** activity from all other activities in the process
2. **status:** the status of the **<invoke>** execution. Before being executed, the **<invoke>** activity is *inactive*. During execution, it has the status of *processing*, and after a successful execution it has the status of *done*. If the service invocation fails, its status is *failed*. The status is *cancelled* when the cancellation service operation is executed as a result of the process rollback, if the complete process is failed.
3. **importance:** the importance of a successfully executed action inside the process. The execution of the action can be either *vital*, if the action is essential for the complete execution of the process, or *negligible*, if the functionality of the process is carried out even without execution of the action. This attribute enables the definition of optional-composite services.

4. **WebServiceInvoked**: the actual Web service endpoint that was used to perform the service operation
5. **invokedBy:** the identification of the software agent that invoked the Web service
6. **invocationTime:** the time, in milliseconds, that the Web service took to be executed
7. **failureCause:** an explanation of what caused the service to fail

Figure 4-2 (above) shows how the level attribute is used to uniquely identify each of the activities within a process.

The same service can be present at more than one level inside the process, such as the Flight Booking service in this example, because each level is uniquely identified inside the process. Figure 4-4 (below) shows an instance of the XML Schema for business processes that defines the travel-service process represented in Figure 4-2.

4.4 Data representation

Every composite service requires a definition of the information that is characteristic of the service action context. Atomic services involved in the travel service share data related to issues in the travel service scenario (departure and return dates, origin and destination, etc.). Thus, software agents involved in executing a specific composite service must manipulate the same kind of information (e.g., software agents involved in the travel service must manipulate information related to travel). A software agent can collaborate in the execution of two different composite services (e.g., the agent that provides the travel insurance service can be involved in a scenario that requires mortgage insurance) but it must be able to manipulate two different classes of information. Therefore, a definition of the data required for each composite service is necessary to facilitate its manipulation by software agents involved in the service execution.

```
<wscp:process  xmlns:wscp=http://mmarl.org/XMLProcess
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://mmarl.org/XMLProcess http://mmarl-021:8080/schemas/process.xsd"
 ID="P0000" name="Travel Service" status="inactive" >
 <wscp:sequence level = "S0" status="inactive" >
  <wscp:flow level = "F1">
   <wscp:invoke level = "I11" importance="negligible" status = "inactive">
    <wscp:service>AttractionsSearch</wscp:service>
   </wscp:invoke>
   <wscp:sequence level = "S12" status  = "inactive">
    <wscp:decision level = "D121" status  = "inactive">
     <wscp:if>
           <wscp:predicate>ItIsAnInternationalFlight</wscp:predicate>
     </wscp:if>
     <wscp:then>
      <wscp:sequence level = "S1211" status = "inactive" >
       <wscp:invoke level="I12111" importance="vital" status="inactive" >
        <wscp:service>FlightBooking</wscp:service>
       </wscp:invoke>
       <wscp:invoke level="I12112" importance="vital" status="inactive" >
        <wscp:service>TravelInsurance</wscp:service>
       </wscp:invoke>
      </wscp:sequence>
     </wscp:then>
     <wscp:else>
      <wscp:invoke level = "I1212" importance="vital" status = "inactive" >
       <wscp:service>FlightBooking</wscp:service>
      </wscp:invoke>
     </wscp:else>
    </wscp:decision>
    <wscp:invoke level = "I122" importance="vital"  status = "inactive">
     <wscp:service>HotelBooking</wscp:service>
    </wscp:invoke>
   </wscp:sequence>
  </wscp:flow>
  <wscp:decision  level = "D2" status = "inactive" validation= "undefined">
   <wscp:if>
             <wscp:predicate>StayDaysAreMoreThan</wscp:predicate>
             <wscp:parameter>3</wscp:parameter>
   </wscp:if>
   <wscp:then>
    <wscp:invoke level = "I21" importance="negligible" status = "inactive">
     <wscp:service>CarRental</wscp:service>
    </wscp:invoke>
   </wscp:then>
  </wscp:decision>
  </wscp:sequence>
</wscp:process>
```

Figure 4-4. Definition of the travel service business process

The data that is required to execute composite services and the business-process definition are handled in different artifacts. The separate manipulation of data and process information offers the following benefits: (i) data can easily be manipulated, stored, and presented to the user without looking at the process information; (ii) the same process modelling language can be used to define different composite services, and (iii) the same code for executing a business process can be used in different contexts of action.

The data involved in the travel service scenario is structured according to the XML Schema shown in Appendix E. The information for the travel service starts with the definition of the compulsory *InputData* element. This element contains the minimum and maximum departure and return dates, as well as the destination. The information resulting from the invocation of the flight booking service is stored in the *outputData* element. A data document is then exchanged among participant agents to create a final document with the information about the complete travel service.

Note that the user information is contained in a separate artifact. It is also exchanged among software agents in the same message that contains the process definition and the data resulting from the service invocation.

4.5 Business process execution

A composite service is modelled as a business process by using an activity diagram. The activity diagram indicates the sequential and concurrent execution of activities, decision points that define the flow of control, as well as the execution of actions. An action represents the invocation of a specific service operation. This section presents how a business process is executed through the collaboration of software agents.

4.5.1 Agents involved in the process execution

The execution of composite services in ASTEK is performed through the collaboration of the following software agents:

1. **Activity-executor agent**: represents atomic services. This agent is responsible for invoking a specific service operation by contacting the Web service that offers the operation that the agent represents, and for collaborating with its peers to execute the entire process. It continuously searches the UDDI registry (through a UDDI agent) to

find Web services that provide the operation that the agent represents. If available Web services are found, then the activity-executor agent registers itself in the Director Facilitator (DF) of the Agent Platform in order to be discovered by its peers. The agent is continuously monitoring Web services by using their ping operation, to see if they are still available. If no Web services are available, the agent deregisters itself from the DF.

2. **Process-broker agent:** represents a specific composite service. When a process-broker agent is created, it searches the process repository to obtain the XML document that describes the composite service that the agent represents. Then, it searches the DF to ensure that all the activity-executor agents involved in the service execution are available. If so, the agent registers itself in the DF to make the composite service available in the system. The agent continuously checks the registration of activity-executor agents; if some of them are unavailable, the agent deregisters itself from the DF. A composite service is available only if the process-broker agent that represents the service is registered in the DF.
3. **User agent:** represents the person that wants to invoke an available service. A user agent is created when a user is logged onto the system. This agent obtains user information from the user database. It provides the user with a graphical interface that contains a list of the available services in the system. The user agent searches the DF for registered process-broker agents to display the list of available services.
4. **Process-executor agent:** initiates the execution of a composite service, receives the resultant information, and presents it to the user. It is created by a process-broker agent for every process-execution request it receives. The process-executor agent

presents a graphical user interface (GUI) to the user to insert the information needed for the service invocation.

5. **UDDI agent:** receives requests to find Web services registered in a UDDI registry. The searching criteria are given by the interface of the Web service and its namespace.

Further details of the implementation of these agents are provided in Section 5.2.

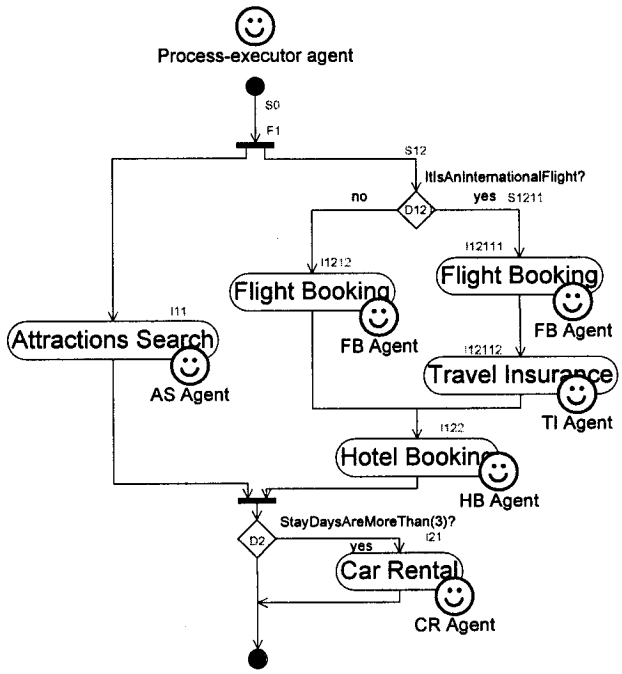


Figure 4-5. Activity-executor and process-executor agents in the travel service

Figure 4-5 depicts how each action in the travel service is represented by an activity-executor agent. Consequently, the system contains the following activity-executor agents: a Flight Booking Agent (FBA), a Hotel Booking Agent (HBA), a Travel Insurance Agent (TIA), a Car Rental Agent (CRA), and an Attractions Search Agent (ASA). The figure also shows the process-executor agent that is in charge of initiating the travel service execution. Note that the flight booking operation appears in actions at two different levels in the diagram; however, the same FBA can perform the action at both levels. Based on the evaluation of

the decision at level D121, the FBA invokes either the action in level I1212 or in level I1211. The FBA then executes the subsequent activities.

4.5.2 Interaction among agents for completing the process execution

The sequence of messages exchanged among the different software agents involved in the service execution is shown in Figure 4-6.

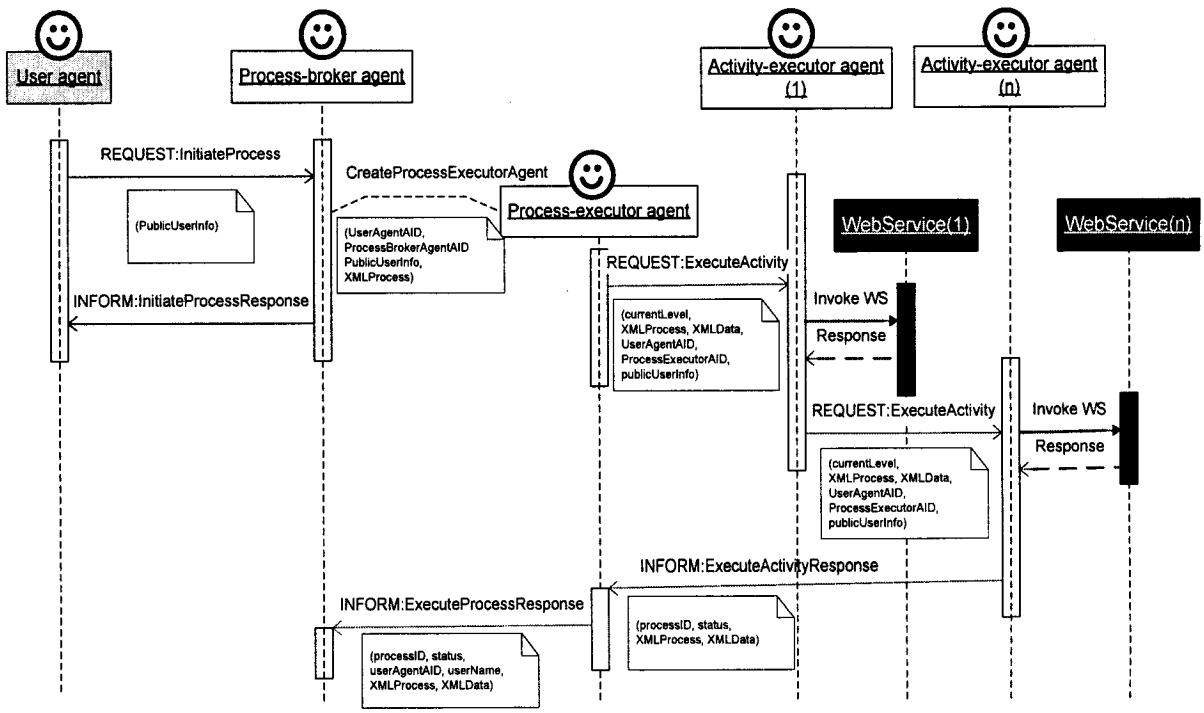


Figure 4-6. Sequence diagram for process execution

The actions for executing a business process are the following:

1. A user agent requests a process-broker agent to execute the service it provides.
   a) The user starts a session in the system by requesting the creation of a user agent. The user agent obtains user information from the user database and displays it in a graphical user interface (GUI).
   b) The user agent shows a list of available composite services; the list contains the services that are offered by process-broker agents registered in the DF.

Therefore, the displayed list will contain the travel service if the process-broker agent that represents it is already registered in the DF.

c) The user selects, from the list of available services, the one to be executed and confirms its execution by clicking a button.

d) The user agent sends an **InitiateProcess** message to the process-broker agent to initiate the process execution. This message contains the public information of the user that is required to invoke the service.

2. The process-broker agent creates a process-executor agent and assigns the process execution to it.

   a) After receiving the **InitiateProcess** message, the process-broker agent creates a copy of the template of the process definition it represents (as shown in Figure 4-4) and assigns to it a new process ID value.

   b) The process-broker agent creates a process-executor agent and provides it with information about the user agent AID, the process-executor AID, the public information of the user, and the process XML document that contains the new ID.

   c) The process-broker agent returns an **InitiateProcessResponse** message to the user agent to inform it of the initiation of the service execution.

3. The process-executor agent starts the process execution by contacting activity-executor agents.

   a) The process-executor agent shows a GUI in order for the user to insert the information required to execute the service. In the travel service, the user inserts the minimum and maximum departure and return dates, and the destination.

   b) After receiving this information, the process-executor agent validates it and creates an instance of the XML Schema that represents the information to be

manipulated during the process execution and inserts the gathered information. In the travel service example, an XML document is created from the XML Schema shown in Appendix E and the inputData element is filled with the inserted information.

c) The process-executor agent reads the process XML document and locates the first activity to be executed. The agent executes the first activity, as described in Section 4.5.4 (below).

d) When the agent locates an <invoke> activity, it retrieves the service operation to be executed from the content of the <service> element. The agent searches the DF for the activity-executor agent that provides the service operation just retrieved and sends an **ExecuteActivity** message to it. The message contains the level of the activity to be executed, the process and data XML documents, the AID of the process-executor agent, and the public information of the user.

4. The activity-executor agent invokes the Web service and collaborates with other agents to complete the process.

   a) Each activity-executor agent executes a specific operation inside the composite service. After receiving an **ExecuteActivity** message, an activity-executor agent selects, from a set of Web services that provide the same operation, the one that best provides the operation based on the current user information. Thus, the FBA can select one Web service for executive users and a different one for student users. If the activity-executor agent needs private user information (e.g., credit card information), then it could collaborates with the user agent to obtain it.

   b) The activity-executor agent reads the process XML document and locates the next activity to execute. It executes the next activity as described in Section 4.5.4.

c) If the agent locates an <invoke> activity, it retrieves the service operation to be executed from the content of the <service> element. The agent searches the DF for the activity-executor agent that provides the service operation just retrieved and sends an **ExecuteActivity** message to it.

d) If there are no more activities to perform, the activity-executor agent returns an **ExecuteActivityResponse** message to the process-executor agent, which then shows the user the final status of the service invocation and sends the detailed information of the travel service invocation to the user's e-mail address.

4.5.3 Definition of ontologies

As mentioned in Section 2.3.4, the use of ontologies and content languages allows agents to communicate. Agents exchange messages that contain information, from a specific domain, that all of them can understand. Therefore, only agents that share the same ontology can communicate with each other. Ontologies define the possible elements within a domain [52]. The definition of an ontology includes the definition of *terms* and *predicates*. Terms are expressions that identify existing entities inside the domain and that agents talk to and reason with. Terms can be *concepts* or *agent actions*. Concepts are structured data that agents can exchange, and agent actions are the actions that agents can perform. Agent actions are the content of FIPA request messages. Predicates are sentences that express the current status existing inside the domain and are the content of FIPA inform messages.

Agent actions define the type of messages for requesting the execution of a specific action. Predicates define the type of messages for responding to a request message, and concepts define the structured data of message contents [52].

All agents shown in Figure 4-6 share a general ontology called a *common* ontology. This ontology has concepts and predicates that are used by all the agents. Figure 4-7 shows a classes diagram with the elements of the common ontology.

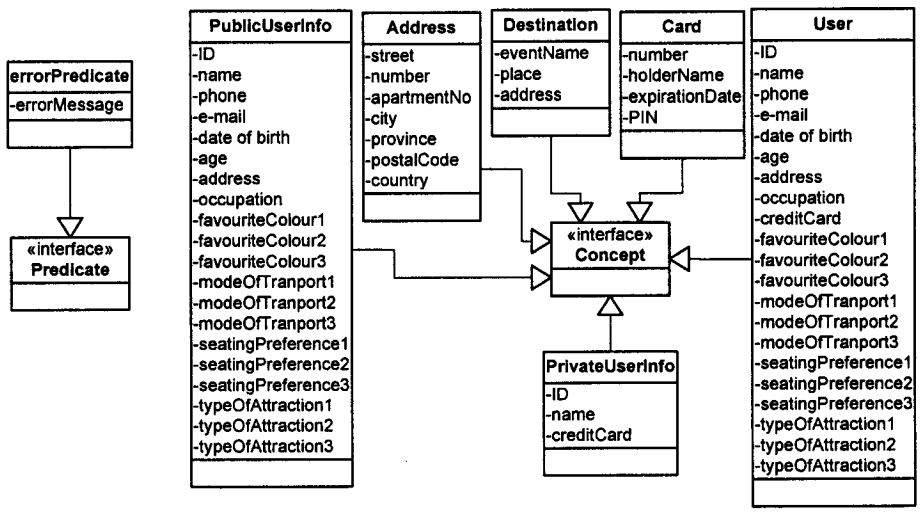


Figure 4-7. Definition of the common ontology

User and process-broker agents can communicate with the messages defined in the *initiateProcess* ontology, which has the elements shown in Figure 4-8(a). Process-broker and process-executor agents share the *executeProcess* ontology, which contains the elements shown in Figure 4-8(b).

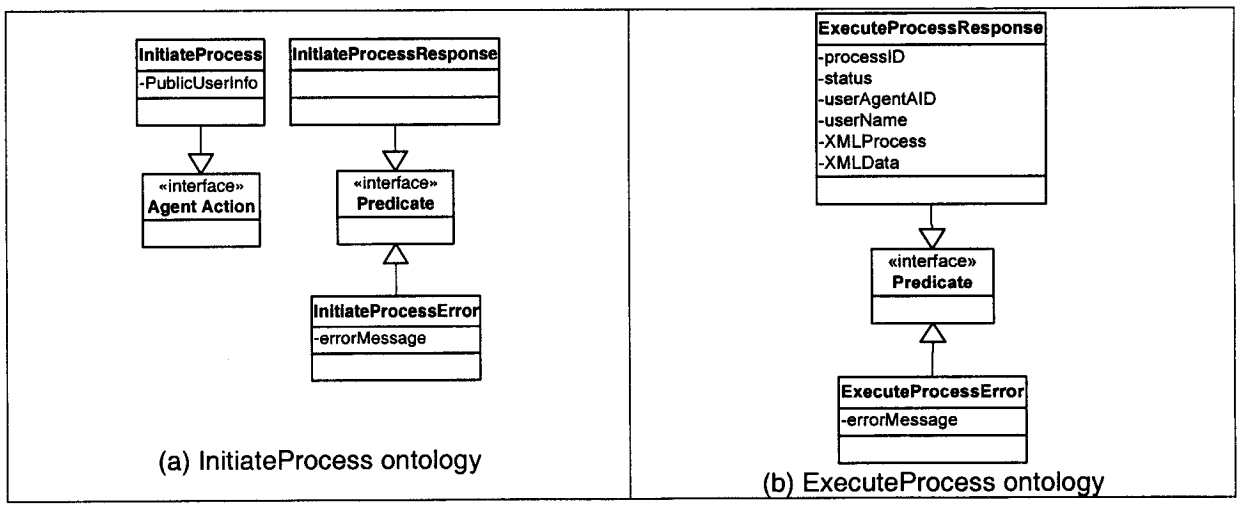


Figure 4-8. Definition of the initiateProcess and executeProcess ontologies

Process-executor and activity-executor agents can communicate by using the messages defined in the *executeActivity* ontology, shown in Figure 4-9.

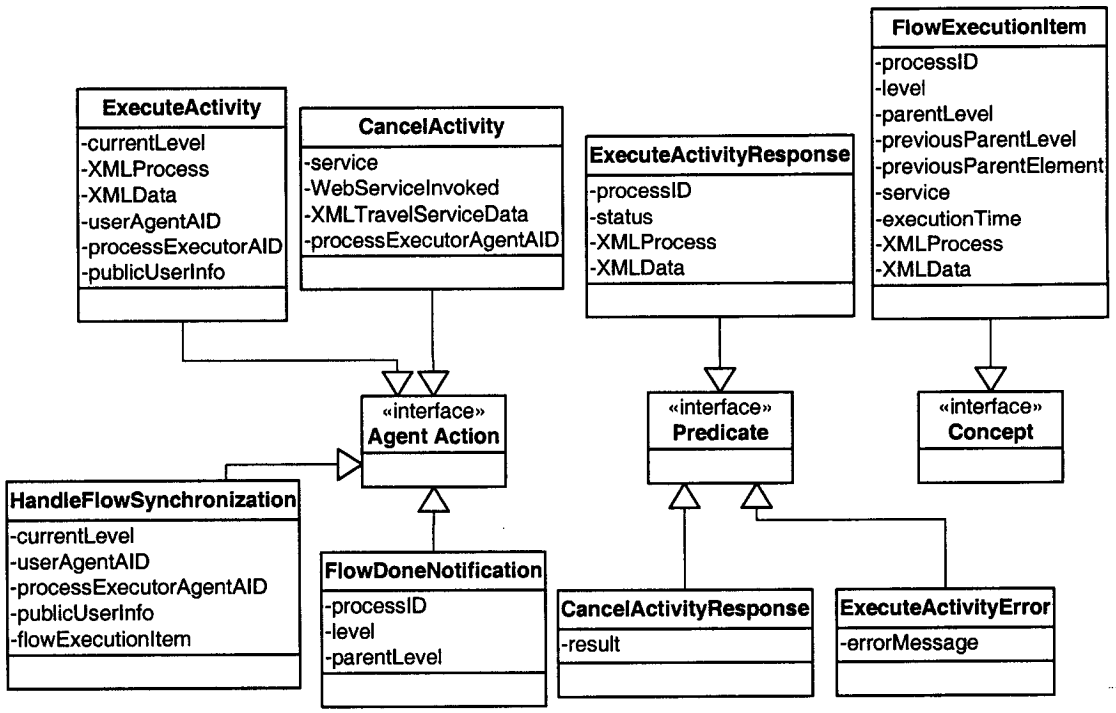


Figure 4-9. Definition of the executeActivity ontology

Activity-executor and UDDI agents can communicate by using the messages defined in the *findWebService* ontology, shown in Figure 4-10.

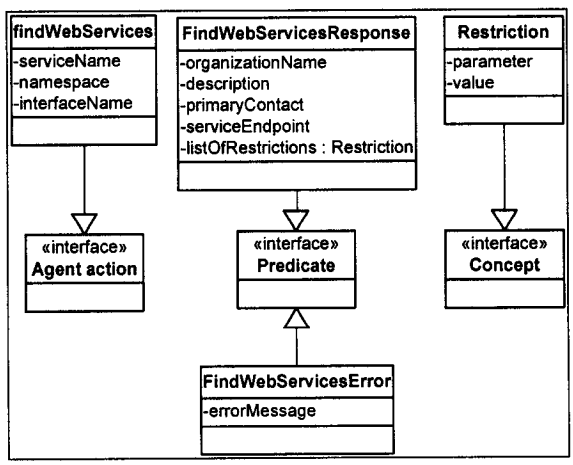


Figure 4-10. Definition of the findWebService ontology

4.5.4 How process-executor and activity-executor agents collaborate to execute a business process

This section describes how a business process is executed as a result of the collaboration of a process-executor agent (PEA) and one or more activity-executor agents (AEAs). As previously explained, there are as many AEAs as available service operations, and one PEA for each business process being executed.

The business process is executed through its process XML document. The PEA and AEAs load the process XML document into their memory and manipulate it by using a tree representation where the root node is the initial <process> element.

4.5.4.1 Invoke activity execution

The simplest business process contains only one <invoke> activity, as shown in Figure 4-11, where the <invoke> activity is in level I1. The process execution is initiated by the PEA, which reads the process XML document and finds the first activity to execute. The PEA finds the <invoke> activity in level I1 and realizes that the service to invoke is *Service1*. Then, it searches the DF for the AEA that provides Service1, and finds that AEA1 provides this service. Next, it sends an **ExecuteActivity** message (a description of this message is found in Figure 4-9) to AEA1 with the *currentLevel* parameter set to I1, and with the updated process and data XML documents.

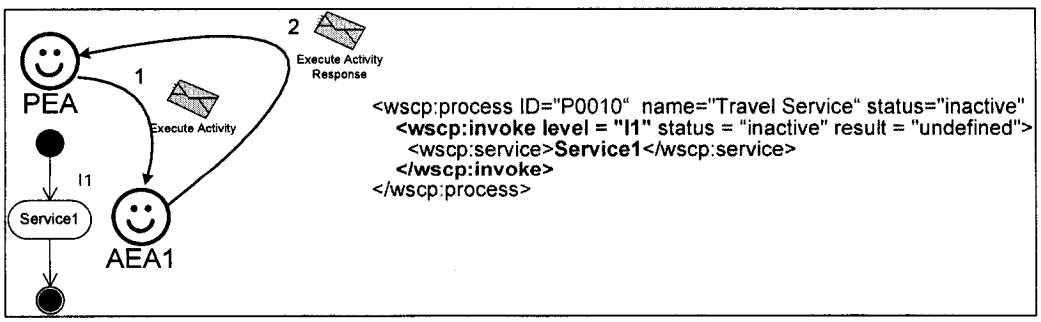


Figure 4-11. Execution of the Invoke activity

AEA1 receives the **ExecuteActivity** message, executes the service, stores the resultant information in the data XML document, modifies the process XML document with information about the process execution, and looks for the parent level of level I1. AEA1 realizes that the parent level of I1 corresponds to the root of the document; therefore, no activities remain to be executed. Then, it sends an **ExecuteActivityResponse** message to the PEA, with the resultant information from the process execution.

4.5.4.2 Sequence activity execution

Figure 4-12 shows a business process that contains two <invoke> activities that are executed in a sequential manner. The two activities are located in levels I1 and I2. The process execution is started by the PEA, which reads the process XML document and finds the first activity to execute. The PEA finds the <sequence> activity in level S0 and then processes the first activity of the sequence. It realizes that the first activity in the sequence is the <invoke> activity in level I1 and that the service to invoke is Service1. Then, it searches the DF for the AEA that provides Service1, and finds that AEA1 provides this service. Next, it sends an **ExecuteActivity** message to AEA1 with the currentLevel parameter set to I1, and with the updated process and data XML documents.

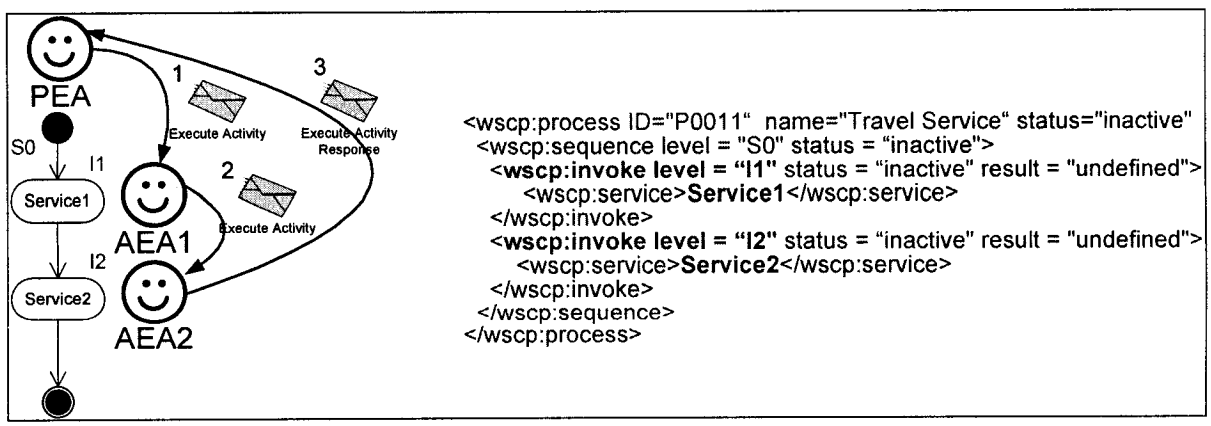


Figure 4-12. Execution of the Sequence activity

AEA1 receives the **ExecuteActivity** message, executes the service, stores the resultant information in the data XML document, modifies the process XML document, and looks for the parent level of level I1. AEA1 realizes that the parent level corresponds to the <sequence> activity in level S0. Then, it searches for the next activity to process in this sequence, finding that the <invoke> activity in level I2 is the next activity, and that the service to invoke is Service2. The PEA searches the DF for the AEA that provides Service2, and finds that AEA2 provides this service. Then AEA1 sends an **ExecuteActivity** message to AEA2 with the currentLevel set to I2, and with the updated process and data XML documents.

AEA2 receives the **ExecuteActivity** message, executes the service, stores the resultant information in the data XML document, modifies the process XML document, and looks for the parent level of level I2. It realizes that the parent level corresponds to the <sequence> activity in level S0. Then it searches for the next activity to execute in this sequence, finding that there are no more activities to process. It looks for the parent of the activity in level S0 and realizes that it is the root of the document. Finally, it sends an **ExecuteActivityResponse** message to the PEA, with the resultant information from the process execution.

4.5.4.3 Decision activity execution

Figure 4-13 (below) shows a business process that contains a <decision> activity in level D0 as the first activity to execute. The <then> element of the decision contains an <invoke> activity in level I1, whereas the <else> element contains a <sequence> activity in level S2. The <sequence> activity contains two <invoke> activities at levels I21 and I22, respectively. The process execution is started by the PEA, which reads the process XML document and seeks the first activity to execute. The PEA finds the <decision> activity in level D0 and then

looks for the predicate to be evaluated. It evaluates the predicate with the current information that it has. If the result of the evaluation is *true*, the next activity to execute is the one in the <then> element; otherwise, the next activity is the one in the <else> element.

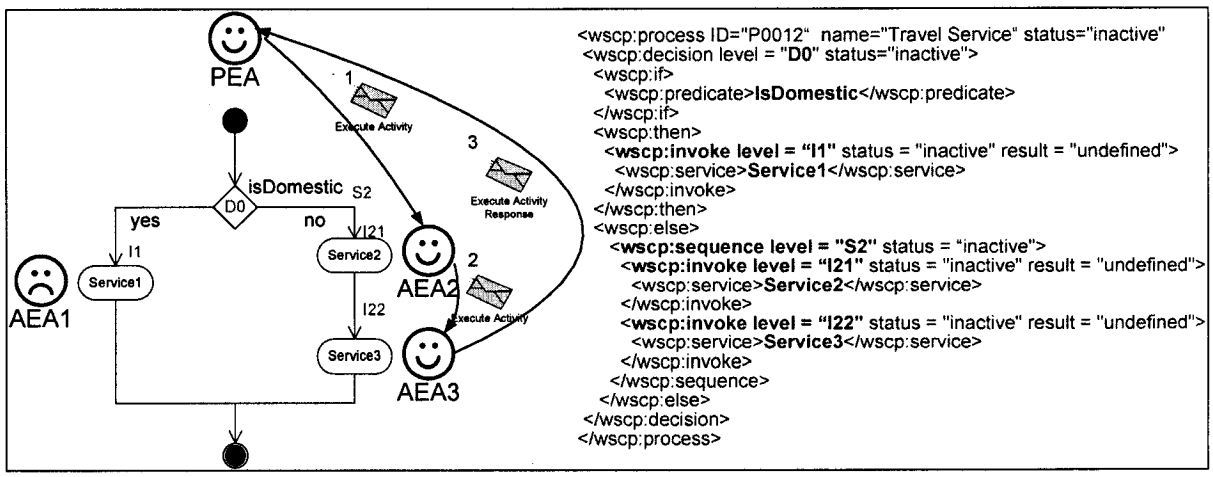


Figure 4-13. Execution of the Decision activity

If the resultant evaluation is *false*, the PEA looks for the activity in the <else> element, which is the sequence in level S2. Let us say that the resultant evaluation is true. Then, the PEA realizes that the first activity in the sequence is the <invoke> activity in level I21, and that the service to invoke is Service2. Then, it searches the DF for the AEA that provides Service2 and finds that AEA2 provides this service. Next, it sends an **ExecuteActivity** message to AEA2 with the currentLevel parameter set at I21, and with the updated process and data XML documents.

AEA2 receives the **ExecuteActivity** message, executes the service, stores the resultant information in the data XML document, modifies the process XML document, and looks for the parent level of level I21. It realizes that the parent level corresponds to the <sequence> activity at level S2. Then, it searches for the next activity to execute in this sequence, finding that the <invoke> activity in level I22 is the next activity to execute and that the service to invoke is Service3. The PEA searches the DF for the AEA that provides Service3 and finds

that AEA3 provides this service. Then, AEA2 sends an **ExecuteActivity** message to AEA3 with the currentLevel set at I22, and with the updated process and data XML documents.

AEA3 receives the **ExecuteActivity** message, executes the service, stores the resultant information in the data XML document, modifies the process XML document, and looks for the parent level of level I22. It realizes that the parent level corresponds to the <sequence> activity at level S2, and that there are no more activities to execute in the S2 sequence. Then, it looks for the parent of the activity in level S2 and realizes that it is the <decision> in level D0. It searches for the parent of D0 and reaches the root of the document. Finally, it sends an **ExecuteActivityResponse** message to the PEA with the resultant information from the process execution.

4.5.4.4 Flow activity execution

Figure 4-14 (below) shows a business process with four <invoke> activities to be simultaneously executed. The <invoke> activities are located in the levels I1, I2, I3, and I4, respectively. The process execution is started by the PEA, which reads the process XML document and looks for the first activity to execute. It finds the <flow> activity in level F0 and then processes each activity in the flow. It realizes that the activities to execute are the <invoke> activities in levels I1, I2, I3, and I4, and that the services to invoke are Service1, Service2, Service3, and Service4. Then, it searches the DF for the AEAs that provide these services and finds that AEA1, AEA2, AEA3, and AEA4 provide these services, respectively. Next, it sends **ExecuteActivity** messages to these AEAs with the currentLevel parameter set at the corresponding level for each <invoke> activity, and with the updated process and data XML documents.

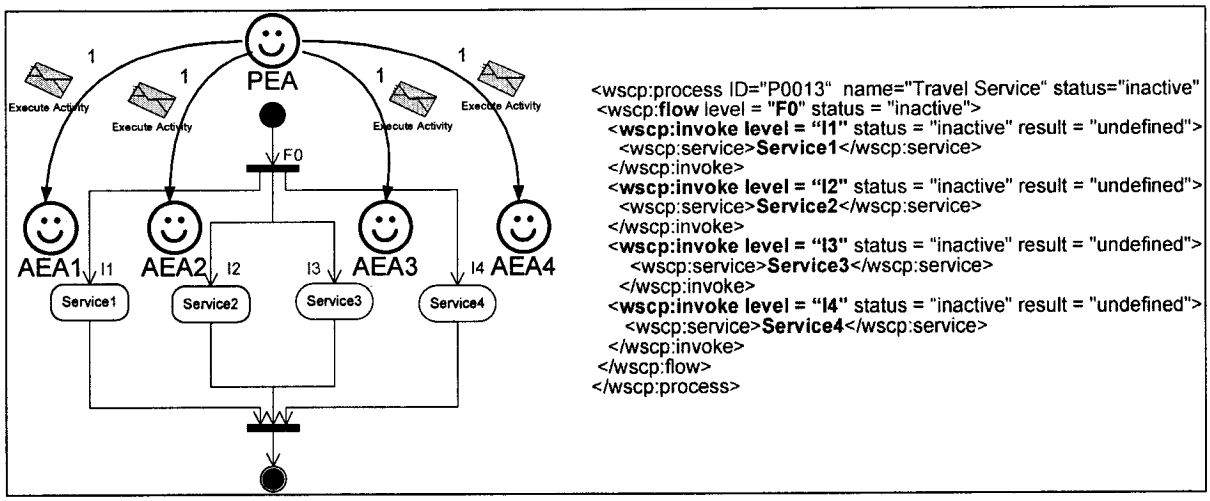


Figure 4-14. Execution of the Flow activity

The AEAs receive the **ExecuteActivity** message, execute the corresponding service, store the resultant information in the data XML document, modify the process XML document with information about the execution of the process, and look for the parent level of the currentLevel received. They realize that the parent level corresponds to the <flow> activity in level F0.

At this point in the flow activity execution, the AEAs participating in the execution of a <flow> activity enter a stage where they share partial information about the execution of the process and elect an AEA to merge the partial information to continue the execution of the process. This stage consists of the following phases: (1) synchronization, (2) leader election, (3) notification, and (4) termination.

4.5.4.4.1 Synchronization phase

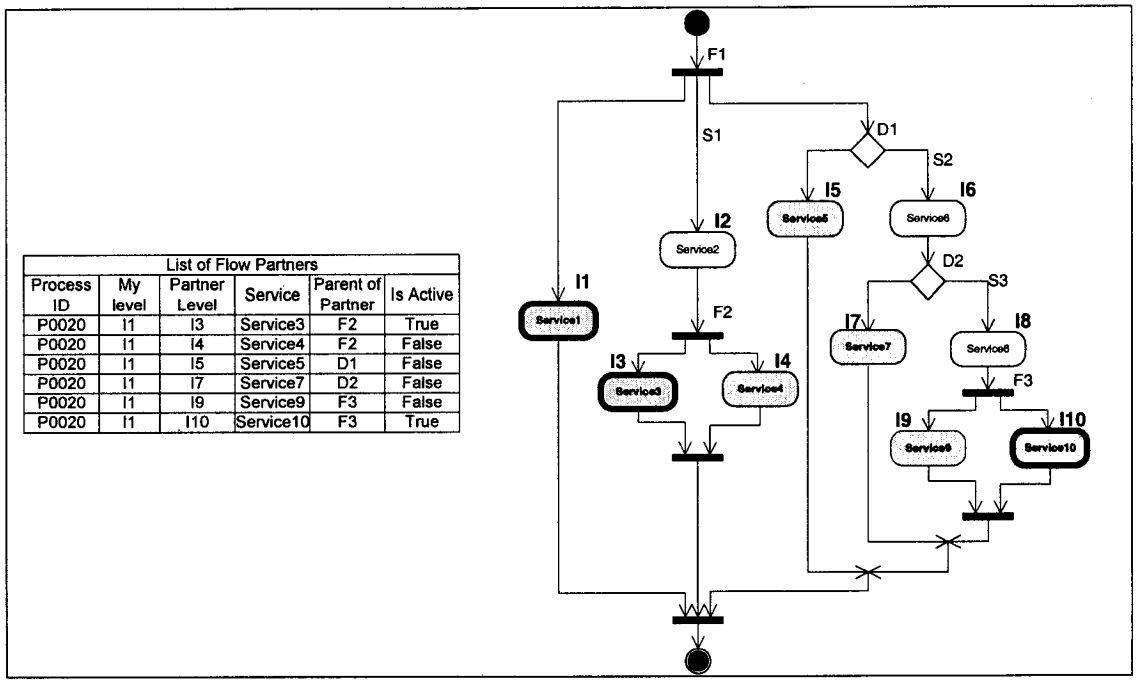
In this phase, activity-executor agents that are performing the activity at the last level of a branch of a <flow> share the partial information of the execution in that branch so that all of them have the same information.

Each activity-executor agent participating in the synchronization phase uses two data structures: a *List of Flow Partners* and a *Flow Execution Buffer.*

List of Flow Partners

A List of Flow Partners (LFP) is created by an AEA when completing the execution of an <invoke> activity whose level *L* is the level of the last activity to be executed in a branch of the level *F* <flow> activity. The LFP contains information about the levels of <invoke> activities in the remaining branches of the level F <flow> activity, which are potentially the last to be executed. These levels are the *partner-levels* of level L. For example, in the business process in Figure 4-14 (above), under the level F0 <flow>, the partner levels of level I1 are I2, I3, and I4.

An AEA stores the following information about its partner levels in the level F flow: (a) the process identifier, which allows an AEA to simultaneously participate in the execution of more than one process; (b) the level of the activity that the AEA is currently executing; (c) the partner level; (d) the service that is invoked in the activity identified by the partner level; (e) the level of the parent of the partner level. This level corresponds to the level of the <decision> or <flow> activity that is the first parent of the partner level; and (f) a Boolean variable to indicate whether the partner level is competing to be the leader in the level F flow. Figure 4-15 shows a more complex business process in which the AEA in level I1 creates its LFP from the level F1 flow. At the beginning, the <invoke> activities that are potentially the last to be executed are the nodes in grey; therefore, these are the partner levels of I1 and the value of the *IsActive* variable is marked true for all of them.



List of Flow Partners					
Process ID	My level	Partner Level	Service	Parent of Partner	Is Active
P0020	I1	I3	Service3	F2	True
P0020	I1	I4	Service4	F2	False
P0020	I1	I5	Service5	D1	False
P0020	I1	I7	Service7	D2	False
P0020	I1	I9	Service9	F3	False
P0020	I1	I10	Service10	F3	True

Figure 4-15. Example of a List of Flow Partners

Flow Execution Buffer

A Flow Execution Buffer (FEB) is a data structure used by AEAs that participate in the synchronization phase of the level F flow. It contains information about the partial execution of the activities in different branches of flow F. Each entry in the FEB is a Flow Execution Item (FEI). A FEI contains the following information: (a) the identifier of the process being executed, which allows agents to simultaneously participate in the execution of distinct processes; (b) the level of the activity just executed; (c) the level of the <decision> or <flow> activity that is the parent of the activity just executed and that is the nearest to the level F flow; (d) the previous parent level, which is the level of the <decision> or <flow> activity that is the immediate parent of the activity just executed; (e) the name of the service just executed; (f) the date and time at which the service was executed; and (g) the data and (h) process XML documents, which contain partial information on the services executed in a branch of the flow activity. Figure 4-16 shows the data structures of the FEB and FEI.

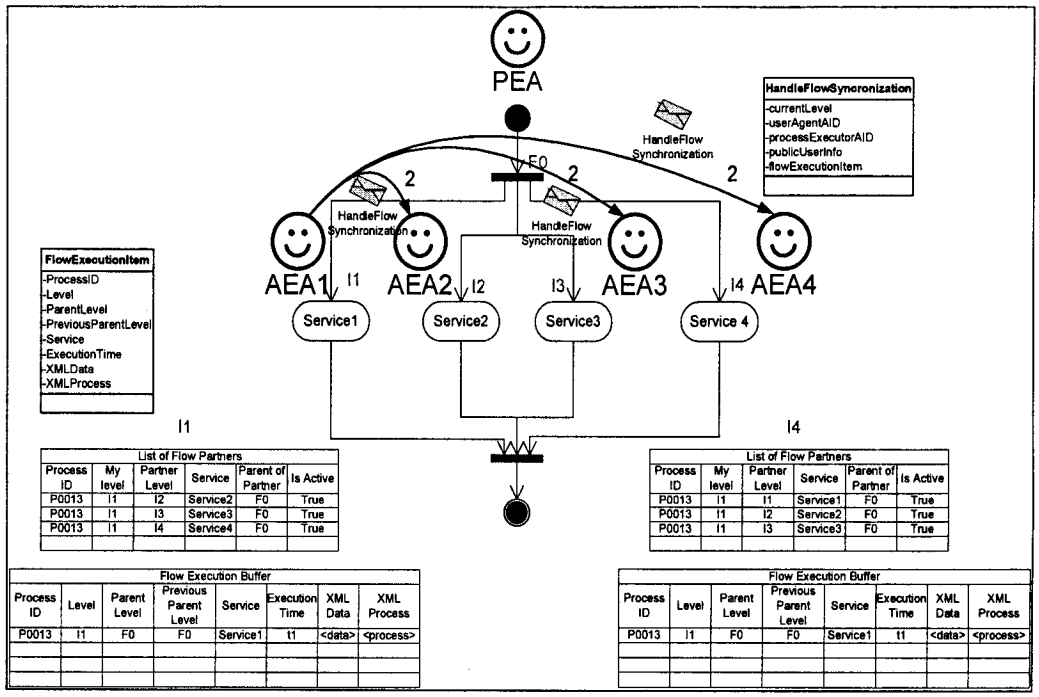


Figure 4-16. The synchronization phase in the execution of a Flow activity

The synchronization phase starts when an AEA completes the execution of the last <invoke> activity in a branch of the level F flow. Then the AEA creates its LFP with the *isActive* field of all its partner levels set at true. Next, it creates an FEI with the partial information on the execution of activities in its particular flow branch, stores the information in its local FEB, and sends a **HandleFlowSynchronization** message (described in the ontology of Figure 4-9) to all AEAs in its partner levels with the isActive field set to *true*. When receiving a **HandleFlowSynchronization** message, AEAs create their LFP (or modify it by changing the isActive field to false in the levels that will not compete to be flow-leaders, if it was previously created), and stores in their FEB the recently received FEI.

Figure 4-16 shows how AEA1 first executes the Service1 and finds that its parent in level I1 is the level F0 <flow>, then creates its LFP and stores the resultant information in its FEB. Next, AEA1 sends **HandleFlowSynchronization** messages to the AEAs in its flow partner levels. The AEAs modify their LFP and store the received information in their FEB.

Eventually, all participant AEAs have the information about the execution of activities in every branch of the level F0 <flow>, as depicted in Figure 4-17.

The number of **HandleFlowSyncronization** messages sent during the synchronization message is $HFS_{messages} = n(n-1)$, where *n* is the number of flow partners in a given flow.

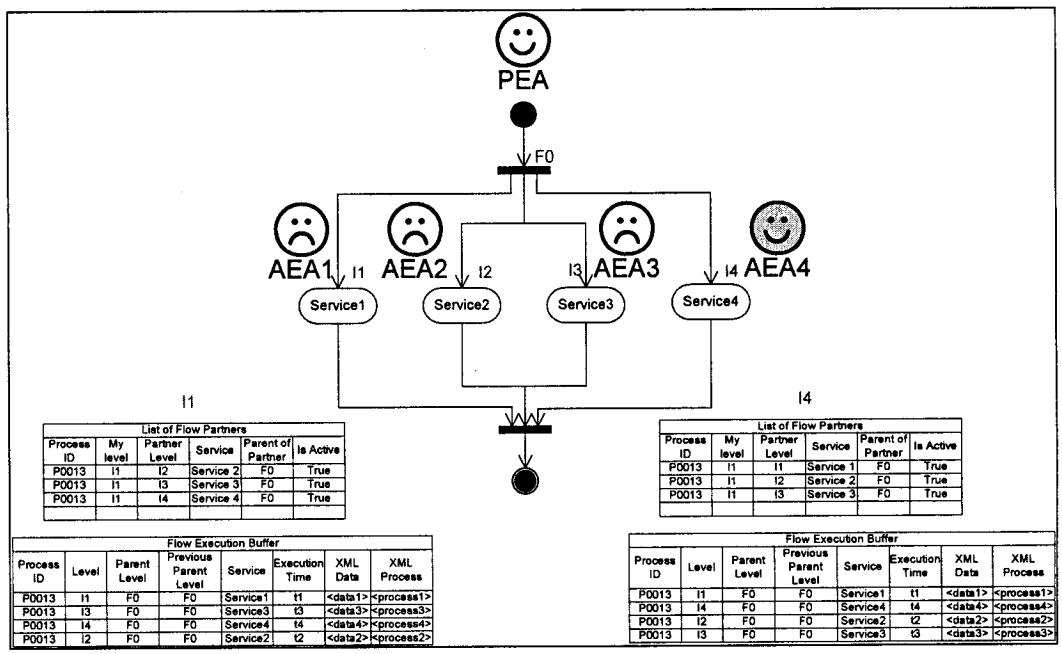


Figure 4-17. Leader election phase in the execution of the Flow activity

4.5.4.4.2 Leader election phase

When a participant AEA contains, in its FEB, information about its own execution and about the execution of activities of its active partner levels, then it can decide whether it is the *flow-leader.*

An AEA in level l_i becomes the flow-leader in the level F_m <flow> if:

(a) $t_i = \max\{t_1,...,t_n\}$, where t_i is the execution time of the service in level l_i, and $t_1,...,t_n$ are the execution times of the *n* flow partners in a given flow. The execution of the service in level l_i was the last one executed in the F_m flow; or

(b) if more than one service is executed at the same maximum time, t_{max}: $hash(l_i) > \max\{hash(l_h), hash(l_i), ..., hash(l_l)\}$, where $l_h, ..., l_l$ are levels in LFP with execution time t_{max}, and $hash(l_x)$ is the hash function that returns an integer value from a string input and has the characteristic that $hash(x) \neq hash(y) \Rightarrow x \neq y$.

A flow-leader is responsible for the following activities: (1) merging the partial data that result from the invocation of services in every branch of the flow activity into a single data XML document; (2) merging the partial process execution information into a single process XML document; (3) sending **FlowDoneNotification** messages to all AEAs with inactive partner levels; (4) cleaning its LFP and FEB; and (4) continuing the process execution. Only the AEAs that are not the flow-leader clean their LFP and FEB data structures.

Figure 4-17 shows that AEA4 is the chosen flow-leader, considering that t_4 is t_{max}.

4.5.4.4.3 Notification phase

In the notification phase, the flow-leader AEA sends **FlowDoneNotification** messages to all AEAs with inactive partner levels in its LFP. Thus, they can clean their LFPs and FEBs. The flow-leader in Figure 4-17 does not have to send notification messages to its flow partners; whereas, if the flow-leader of Figure 4-15 is I1, it must send notification messages to the AEAs in levels I4, I5, I7, and I9.

4.5.4.4.4 Termination phase

In the termination phase, the AEA flow-leader looks for the parent of the <flow> activity it just executed and processes it. In Figure 4-18, the flow-leader AEA4 realizes that its parent is the level F0 <flow>, whose parent is the root of the document. Then, AEA4 sends an **ExecuteActivityResponse** message to the PEA with the resultant information of the process execution.

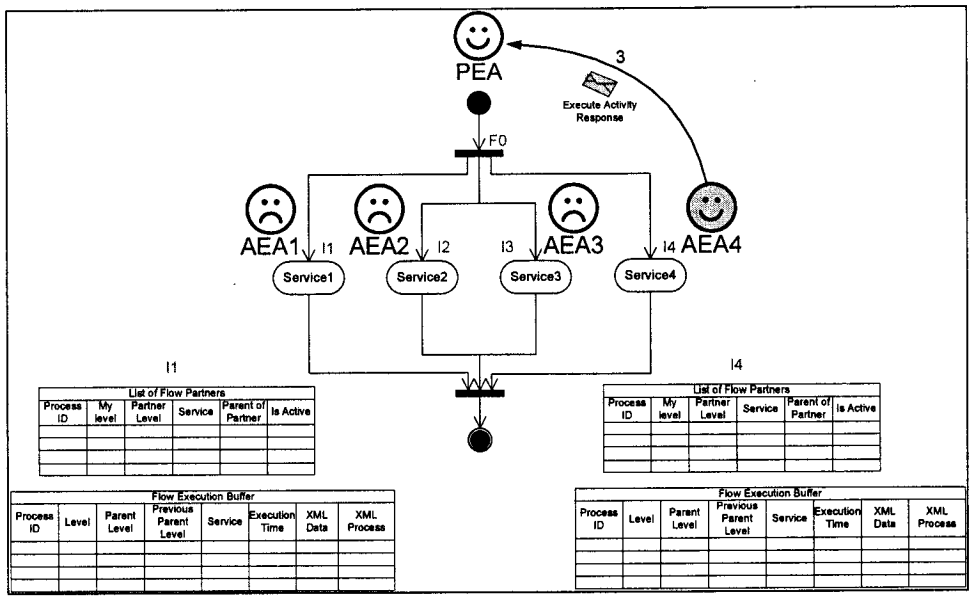


Figure 4-18. Termination phase in the execution of the Flow activity

More complex business processes are executed by recursively using the previously described activity-execution algorithms.

4.5.5 Fault-recovery support

ASTEK provides support for handling faulty service invocations. A service invocation can fail in two situations: (a) if the preconditions to invoke the service are not fulfilled; and (b) if the invocation of the Web service fails (e.g., interconnectivity problems, denial of service, or use of an incorrect service endpoint address).

When an activity-executor agent detects a faulty service invocation, it takes into account the *importance* attribute of the <invoke> activity to determine whether the entire process is also faulty. If the importance parameter is *vital*, the AEA sets the status of both the <invoke> and <process> activities as failed. Then, it looks for the parent level of the failed activity. If the parent of the faulty activity is a <flow> element, the AEA executes the flow activity, exchanging the faulty process XML document with its flow partners. Eventually one of them will be the flow-leader and it will merge the contents of the partial process XML documents;

however, the final process XML document is marked as failed. When the traversal reaches the root of the document, the AEA sends the process and data XML documents to the PEA. The PEA realizes that the process is faulty and sends **CancelActivity** messages to all the AEAs that successfully invoked an atomic service in the faulty process, in order to invoke the cancellation operation of the invoked Web services.

If the service invocation is *negligible*, only the <invoke> activity is marked as failed and the process is executed as if nothing happened.

4.5.6 Service discovery

Service discovery is necessary in two situations: (a) when process-executor and activity-executor agents search for activity-executor agents that provide a specific service operation, and (b) when activity-executor agents search for Web services that offer a specific service operation. In the first scenario, agents use the agent platform's Director Facilitator (DF) to search for activity-executor agents according to the services they provide. Activity-executor agents are registered in the DF only if they can provide a service. If an activity-executor agent cannot contact any Web service that provides the service that the agent represents, the activity-executor agent deregisters itself from the DF. It will register itself again once it can detect an available Web service. The service information that is stored in the DF contains the activity-executor agent AID, the service operation it provides, the ontologies it handles, and information about the Web services it can contact (i.e., the endpoint address where the Web service can be invoked).

In the second scenario, activity-executor agents search the UDDI service repository to discover Web services that provide the operation they represent. An activity-executor agent does not need to discover Web services if the service operation it represents is executed by a particular Web service, such as when only one provider offers this service. In this case,

the Web service endpoint address is hard-coded within the activity-executor agent that provides the service. If the service operation is provided by more than one Web service, the AEA searches the UDDI registry to find the available Web services with the help of a UDDI agent. Figure 4-19 depicts the interaction between activity-executor and UDDI agents.

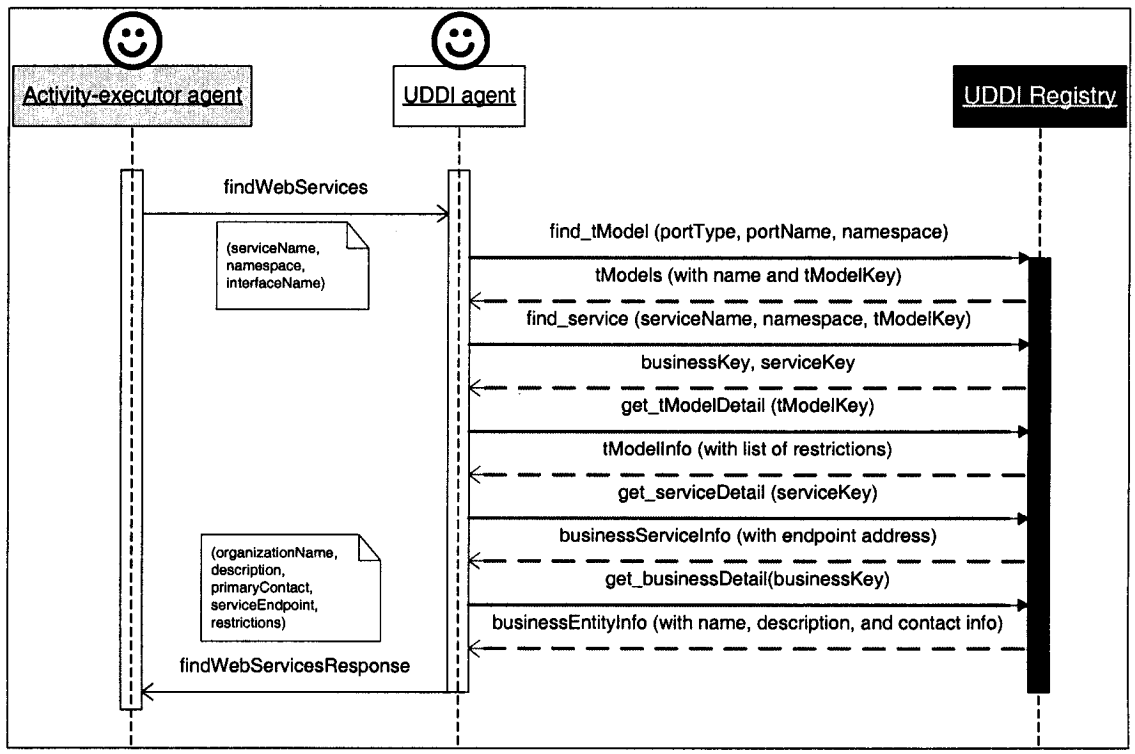


Figure 4-19. Service discovery using a UDDI agent

When an AEA needs to discover Web services that provide the service operation it represents, it sends a **findWebService** request message to the UDDI agent. The message contains the name of the service, the namespace that identifies the type of service, and the name of the service interface. The UDDI agent uses this information to contact the UDDI registry and to find Web services that offer the service with a given interface and namespace. The UDDI agent returns a response message that contains a list of the Web services that provide such a service. The list contains the provider's name, a description of the provider, the primary contact information, the endpoint address where the service is

invoked, and the restrictions for invoking the service (e.g., the type of users that can access the service, such as executive users). Details about the requirements for registering Web services in a UDDI registry are in Section 5.4.

4.6 ASTEK and Web service composition problems

This subsection presents how ASTEK solves the problems associated with Web service composition, which were discussed in Section 3.2.

4.6.1 Service orchestration

ASTEK implements the reactive service-orchestration approach: Web services are selected and integrated into the composition at execution time. ASTEK uses an activity diagram to represent the service operations involved in a business process and, at execution time, a Web service is selected from a set of Web services offering the same operation.

4.6.2 Service choreography

With ASTEK, the service choreography to execute a business process is based on the cooperation between one process-executor agent, which is in charge of initiating the process execution, and one or more activity-executor agents, which invoke the Web services and collaborate with their peers to execute the process.

4.6.3 Service discovery

Using ASTEK, service discovery is necessary in two situations: when searching for activity-executor agents that provide a specific service operation in a given agent platform, and when searching for Web services that provide a specific service. In the first situation, activity-executor agents are registered in the agent platform's DF to be discovered by their peers and to enable the service composition. In the second situation, activity-executor

agents contact the UDDI agent, which directly interacts with a UDDI registry, to search for available Web services offering the service operation that the activity-executor agents represent. Service providers follow certain public recommendations to map the WSDL documents that describe their services to UDDI entries to enhance dynamic service discovery. Service providers must also modify the service registration when service-invocation conditions change, for example, when services become unavailable.

4.6.4 Level of scalability and flexibility

ASTEK provides a high level of scalability. Service providers register, in the same UDDI registry, their Web services that are to be integrated into the service composition process. Web services that provide the same functionality use an agreed-upon service interface and are described with a WSDL document and an agreed-upon namespace. Thus, the UDDI registry congregates a large number of Web services that offer the same functionality.

To enhance the service discovery process, service providers must take into account the agreed-upon service interface and namespace when publishing a Web service. They must also follow some public recommendations to map the content of WSDL elements into UDDI entries. The registration of a Web service in the UDDI registry also contains some parameters that differentiate the use of the Web services in particular situations, such as when the service is only intended to be used by users that are catalogued as executives, in the travel service scenario.

ASTEK offers a high level of flexibility. An activity-executor agent receives a list of available Web services that offer the same functionality through the same interface. Then it selects the one that matches the restrictions of its use based on the user information or intermediate data that resulted from the invocation of previous Web services. If no Web services match the restrictions, a default Web service is invoked.

4.6.5 Service communication

In ASTEK, service invocation is performed by using request-response messages (remote method invocations).

4.6.6 Recursiveness

ASTEK does not provide a WSDL interface to make the composite services provided within the framework available. This could be implemented in future work.

4.6.7 Definition of interfaces

The definition of service interfaces in ASTEK is important because Web services that offer the same functionality can be substitutable used only if they use the same interface and are described with the same namespace. An activity-executor agent recovers the endpoint address of the selected Web service and uses the same code to invoke the Web service using the agreed-upon service interface.

4.6.8 Participation of atomic services

ASTEK considers both negligible and vital atomic services. Therefore, optional-composite services, in which the successful execution of the entire composite service is completed even with the failure of negligible atomic services, can be created.

4.6.9 Fault-recovery support

ASTEK can handle faulty service executions in which Web services already invoked in the faulty composite service execute a cancellation operation to roll back the actions already performed.

4.6.10 Data representation

ASTEK uses two different artifacts to allocate intermediate data and control-flow information. Intermediate data is stored in an XML document by activity-executor agents after they successfully execute a Web service and receive the data resulting from the service invocation. All information resulting from the service execution is stored in another XML document. Both documents are exchanged among agents as the content of a single message during the service execution.

The data representation depends on the specific composite service. This thesis uses the travel service scenario. Implementation of another scenario would require a different data representation.

4.7 Summary

This chapter presented the design of an Agent-based Web Service composiTion framEworK (ASTEK) that uses standard Web service technologies to offer a service composition approach with high levels of scalability and flexibility. It reduces the extra work that service providers must perform to integrate their services into the service composition process.

First, this chapter introduced ASTEK and the travel service scenario, which is the software application that benefits from the service composition. Then, the language that defines the composite services within ASTEK was described. Next, the data representation that is used in the travel service scenario for composing services within ASTEK was also described. The protocol that software agents follow for composing services was detailed. Finally, the way in which ASTEK solves the problems related to Web service composition was explained.

Chapter 5
Implementation of the Agent-Based Web Service Composition Framework

This chapter presents the implementation of the Agent-based Web Service composiTion framEworK (ASTEK) as follows. Section 5.1 introduces the architecture of ASTEK describing the elements that comprise the framework as well as the software tools, databases, and software agents inside the framework. Section 5.2 describes the implementation of software agents. Section 5.3 explains the procedure for invoking Web services within ASTEK. Section 5.4 shows how the registration and discovery phases are performed within ASTEK. Section 5.5 describes how Web services can be implemented. Finally, Section 5.6 summarizes the concepts presented in this chapter.

5.1 ASTEK architecture

Figure 5-1 depicts the elements that comprise the ASTEK architecture. Within ASTEK, all software entities are software agents. The agents are deployed in the Java Agent DEvelopment Framework (JADE) [53], which is a FIPA-compliant agent platform implemented in Java language. JADE is deployed in the JAVA Runtime Environment; thus, ASTEK is completely based on Java technologies. JADE incorporates three special agents: the Director Facilitator (DF), the Agent Management System (AMS), and the Remote Monitoring Agent (RMA), which shows a GUI that controls the agent platform and all registered agents [54].

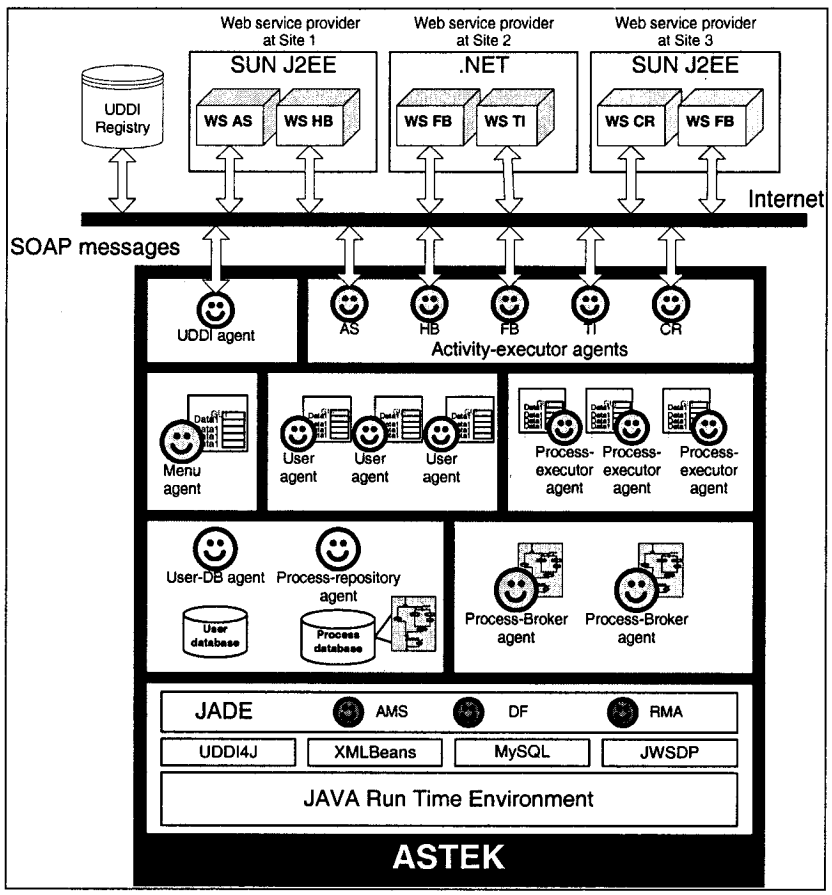


Figure 5-1. ASTEK architecture

5.1.1 Software tools

The following products are used to implement ASTEK:

1. MySQL, an opensource database [55] that provides tools to implement and manipulate databases.
2. XMLBeans [56], a tool for manipulating XML documents as if they were Java classes. This tool incorporates *Cursors* to enhance traversal in XML documents. It is used by agents to manipulate data or process XML documents.

3. Java Web Service Developer Pack (JWSDP) [57], a toolkit that contains the J2EE 1.4 APIs for building, testing and deploying Web services. Agents in charge of invoking Web services use this toolkit.
4. UDDI4J [58], an API for contacting UDDI registries. The agent in charge of service discovery uses this tool.

5.1.2 Databases in ASTEK

ASTEK uses two databases: a user database and a process-repository database.

5.1.2.1 User database

The user database stores information about the users than can invoke the services available in ASTEK. The database contains a table with the following fields: user ID, user name, occupation, address, phone, e-mail, date of birth, and personal preferences.

5.1.2.2 Process database

The process database stores information about the composite services available in the framework and their execution details. The database contains two tables: *processRepository* and *processExecution.*

The processRepository table contains two fields: *processName*, which is the composite-service name, and *XMLProcess*, which is the XML document that describes the composite service. Each composite service available in the framework has an entry in this table.

The processExecution table stores the resultant information from the composite-service execution. An entry on this table is created when a user requests the execution of a composite service. After the service execution is complete, the details of it are stored in the corresponding entry. The processExecution table contains the following fields:

- ProcessID: a consecutive number that identifies the service that is being executed;

- ProcessName: the name of the service being executed (e.g., travel service);
- Status: the phase that the service execution is in. The status is *initiated* when a new entry is created, *done* if the service is successfully executed, or *failed* it is not successfully executed;
- UserID: the ID of the user that invokes the service;
- XMLProcess: the XML document that contains the details of the service execution;
- XMLData: the XML document that contains the resultant data from the service execution;
- CreationDate: the date and time when the entry is created;
- ProcessingTime: the number of milliseconds that the process takes to be executed;
- StartingDate: the date and time when the service starts its execution; and
- FinishingTime: the date and time when the service terminates its execution.

5.1.3 Software agents

As shown in Figure 5-1, ASTEK is composed of different kinds of software agents: (1) the *user-DB* agent, which directly accesses the user database; (2) the *process-repository* agent, which manages the process database; (3) the *process-broker* agent, which represents a composite service registered in the process database; (4) the *menu* agent, which presents a graphical interface for logging onto the framework; (5) the *user* agent, which represents the framework user; (6) the *process-executor* agent, which initiates the process execution; (7) the *activity-executor* agent, which invokes the atomic Web services and collaborates with its peers for the business-process execution; and (8) the *UDDI* agent, which accesses the UDDI service registry to look for available Web services to be integrated into the service composition.

The implementation details of these agents are presented in the following section.

5.2 Implementation of software agents

The agents that comprise the ASTEK framework are JADE agents. They exchange ACL messages whose content is expressed with the FIPA Semantic-Language (SL) content language [60]. JADE agents perform tasks that are modelled and implemented as *behaviour* objects [59]. These behaviours are added to the list of tasks that each agent has and are concurrently executed. Some agents use the FIPA-Request Iteration Protocol that is implemented by using the SimpleAchieveREInitiator and the Responder behaviours [59].

The following subsections describe the implementation of the agents that comprise the ASTEK framework. The description considers the ontologies and behaviours that they use.

5.2.1 User-DB agent

The user-DB agent directly accesses the user database. It can communicate with menu and user agents. A menu agent contacts the user-DB agent when it needs a list of all registered users. A user agent contacts the User-DB agent when it needs to obtain or modify the information of a specific user. The user-DB agent uses the common and userDB ontologies. Its actions are defined by the behaviour described in Table 5.1.

Table 5.1. User-DB agent behaviours

Name: DBResponder	**Class: AchieveREResponder**
This behaviour is added to the task list when the agent starts its execution, and is active while the agent is alive. The agent receives and processes the following messages:	
Message received	**Action performed**
GetUserInfo	Obtains information on a user with a given ID from the user database, and returns the information in a GetUserInfoResponse message.
UpdateUserInfo	Updates the information of the given user with the new information contained in the message. Returns an UpdateUserInfoResponse confirmation message.
GetListofUsers	Obtains from the user database the name and ID of all registered users and returns a GetListOfUsersResponse message with a list of them.

5.2.2 Process-repository agent

The process-repository agent directly accesses the process database. It can communicate with process-broker agents. This agent uses the processDB ontology. Its actions are defined by the behaviour described in Table 5.2.

Table 5.2. Process-repository agent behaviours

Name: DBResponder	**Class: AchieveREResponder**
This behaviour is added to the task list when the agent starts its execution, and is active while the agent is alive. The agent receives and processes the following messages.	
Message received	**Action performed**
GetListofProcesses	Accesses the processRepository table and obtains a list of available composite services. Returns the list in a GetListOfProcessesResponse message.
GetProcessInfo	Obtains the XML document that defines the given process and returns it in a GetProcessInfoResponse message.
UpdateProcessInfo	Updates the definition of a composite service that is stored in the processRepository table.
CreateProcess-ExecutionEntry	Creates an entry in the processExecution table for a new service execution. Returns a CreateProcessExecutionEntryResponse message with a new process ID.
UpdateProcess-ExecutionEntry	Modifies the content of an entry in the processExecution table with the resultant information from the execution of a process with a given process ID.

5.2.3 Process-broker agent

A process-broker agent represents a process registered in the processRepository table of the process database. This agent receives requests from user agents to initiate the execution of the service it represents, creates a process-executor agent for each request received, and waits for confirmation of the process execution. This agent uses the common, processDB, initiateProcess, and executeProcess ontologies. Its actions are defined by the behaviours described in Table 5.3.

5.2.4 Menu agent

The menu agent presents a GUI to create user agents, as shown in Figure 5-2 (below).

Table 5.3. Process-broker agent behaviours

Name: GetProcessInfo	**Class: SimpleAchieveREInitiator**
This behaviour is added to the process-broker agent task list when the agent starts its execution. The agent sends a GetProcessInfo message to the processRepository agent to obtain the XML document that defines the service that the agent represents. The agent waits for a GetProcessInfoResponse message response and, when it receives the message, this behaviour is completed.	
Message received	**Action performed**
GetProcessInfoResponse	Obtains the XML document that defines the service that the agent represents and activates the CheckAtomicServices, InitiateProcessResponder and HandleExecuteProcessResponse behaviours.

Name: InitiateProcessResponder	**Class: AchieveREResponder**
This behaviour is added after receiving a GetProcessInfoResponse message and remains active while the agent is alive. The agent receives and processes the following messages from the user agent.	
Message received	**Action performed**
InitiateProcess	Recovers the information of the user that requests the service execution and activates the GetNewProcessExecutionEntry behaviour to obtain a new process ID. Then, it returns an InitiateProcessResponse message with a request confirmation.

Name: GetNewProcessExecutionEntry	**Class: SimpleAchieveREInitiator**
This behaviour is added to the process-broker agent task list when this agent receives an InitiateProcess message. The agent sends a CreateProcessExecutionEntry message to the process-Repository agent to create a new entry in the processExecution table. The process-broker agent waits for a CreateProcessExecutionEntry-Response message and, when it receives the message, this behaviour is completed.	
Message received	**Action performed**
CreateProcess-ExecutionEntryResponse	Creates a new process-executor agent that presents a user interface to initiate the process execution. The process-executor agent is created with the information about the userAgentAID, the process-BrokerAID, the publicUser-Information, and the XML document of the service to be executed with a new process ID.

Name: HandleExecuteProcessResponse	**Class: CyclicBehaviour**
This behaviour is added to the task list when the process-broker agent receives a GetProcessInfoResponse message, and remains active while the agent is alive. The process-broker agent waits for ExecuteProcessResponse messages from the process-executor agent that it previously created and that has completed the process execution.	
Message received	**Action performed**
ExecuteProcessResponse	Obtains the resultant information from a process execution and activates the UpdateProcessExecutionEntryBehaviour to modify the process-execution entry of the process just executed.

Name: UpdateProcessExecutionEntry	**Class: SimpleAchieveREInitiator**
This behaviour is added to the process-broker agent task list when this agent receives an ExecuteProcess-Response message. The agent sends an UpdateProcessExecutionEntry message to the process-Repository agent to modify the information on the executed process in the processExecution table. The agent waits for an UpdateProcessExecutionEntryResponse message and, when it receives the message, this behaviour is completed.	
Message received	**Action performed**
UpdateProcessExecutionEntryResponse	None.

Name: CheckAtomicServices	**Class: TickerBehaviour**
This behaviour is added to the process-broker task list when this agent receives a GetProcessInfoResponse message and remains active while the agent is alive. Every 5 seconds the agent checks whether the atomic services of the composite service it represents are registered in the DF. Only if all of them are registered will the process-broker agent be registered in the DF, making the service it offers available.	

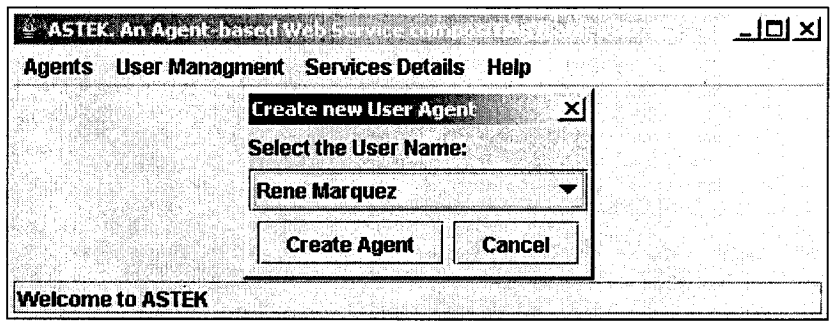
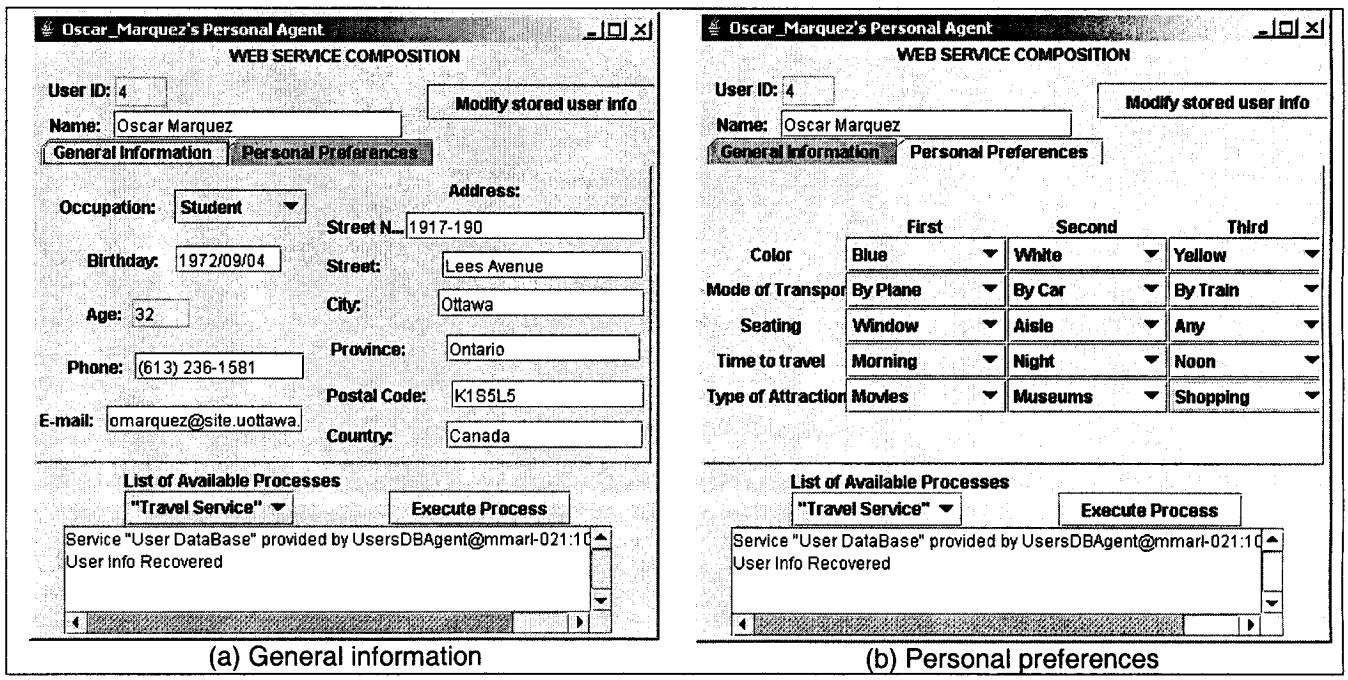


Figure 5-2. Graphical interface of the menu agent

5.2.5 User agent

A user agent represents a user inside ASTEK. This agent is created from the GUI presented by the menu agent. The agent presents a GUI where users can see and modify their own information and select, from a list of available services, the one to be invoked. An example of the GUI presented for the travel service scenario is shown in Figure 5-3. This agent uses the common, initiateProcess, and userDB ontologies. Its actions are defined by the behaviours described in Table 5.4 (below).



(a) General information (b) Personal preferences

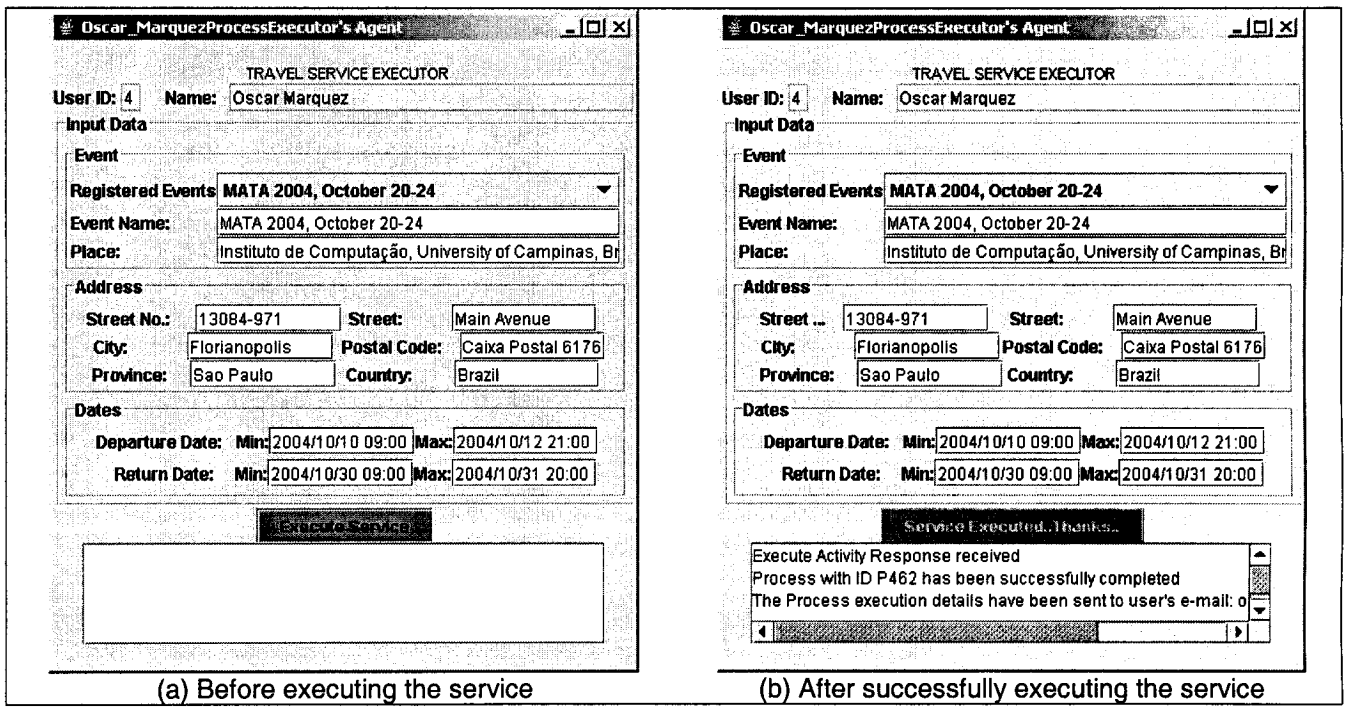
Figure 5-3. Graphical interface of the user agent

Table 5.4. User agent behaviours

Name: GetUserInfo	**Class: SimpleAchieveREInitiator**
This behaviour is added to the task list when the agent starts its execution. The agent sends a GetUserInfo message to the user-DB agent to obtain information on the user it represents. The user agent waits for a GetUserInfoResponse message and, when it receives the message, this behaviour is completed.	
Message received	**Action performed**
GetUserInfoResponse	Obtains information on the user that the agent represents and displays it in a GUI. The agent then searches the DF for available composite services and shows them to the user.

Name: InitiateProcessExecution	**Class: SimpleAchieveREInitiator**
This behaviour is added when the user selects an available service and presses the *Execute process* button. The agent sends an InitiateProcess message to the process-broker agent that represents the selected service to initiate the service execution. The user agent waits for an InitiateProcessResponse message and, when it receives the message, this behaviour is completed.	
Message received	**Action performed**
InitiateProcessResponse	Displays a confirmation message to the user.

Name: UpdateUserInfo	**Class: SimpleAchieveREInitiator**
This behaviour is added when the user presses the *Modify-stored-user-info* button. The user agent sends an UpdateUserInfo message to the user-DB agent to modify the user information with the data contained in the GUI. The user agent waits for an UpdateUserInfoResponse message and, when it receives the message, this behaviour is completed.	
Message received	**Action performed**
UpdateUserInfoResponse	Displays a confirmation message to the user.



(a) Before executing the service (b) After successfully executing the service

Figure 5-4. Graphical interface of the travel process-executor agent

5.2.6 Process-executor agent

A process-executor agent initiates the execution of a composite service. It presents a GUI that enables the user to insert the information required for the service execution. An example of the GUI for the travel service is shown in Figure 5-4. This agent is created by the process-broker agent that represents the service it executes. It searches for atomic services involved in the service execution to activate the *Execute-Service* button, which the user presses to initiate the process execution. Once the agent receives the resultant information from the service execution, it sends an e-mail with the details to the user. An example of the e-mail sent after a successful execution of the travel service is shown in Appendix F. This agent uses the common, executeActivity, and executeProcess ontologies. The actions that this agent executes are defined by the behaviours described in Table 5.5.

Table 5.5. Process-executor agent behaviours

Name: CheckAtomicServices	**Class: TickerBehaviour**
This behaviour is added to the task list when the process-executor agent starts its execution. Every 5 seconds this agent checks whether the atomic services of the composite service it represents are registered in the DF. If all of them are published, the Execute-Service button is activated to allow the user to start the service execution; otherwise, this button is deactivated. This behaviour is active until the user starts the service execution by pressing the Execute-Service button.	
Name: HandleExecuteActivityResponse	**Class: CyclicBehaviour**
This behaviour is added when the user presses the Execute-Service button. This agent sends ExecuteActivity messages to activity-executor agents as a result of the process execution, as described on Section 4.5.4. The process-executor agent waits for an ExecuteActivityResponse message, and when it receives the message, the behaviour is completed.	
Message received	**Action performed**
ExecuteActivityResponse	Recovers the information from the process execution. If the process execution failed, the process-executor agent sends CancelActivity messages to all the activity-agents that successfully executed an atomic service in the failed process. This way, the cancellation operation of the already executed Web services is executed. Then, the process-executor agent waits for CancelActivityResponse messages. Regardless of whether the process execution failed, the agent sends an e-mail to the user with a confirmation of the service execution. The process-executor agent also notifies the process-broker agent that created it of the process execution result by sending it an ExecuteProcessResponse message.
CancelActivityResponse	Displays the message content.

Table 5.6. Activity-executor agent behaviours

Name: SearchUDDIforAvailableWebServices	**Class: CyclicBehaviour**
This behaviour is added to the task list when the agent starts its execution. Every 5 minutes the agent sends a findWebServices message to the UDDI agent, which searches the UDDI registry for Web services that offer the service. Then, the AEA waits for findWebServicesResponse messages. This behaviour is active while the agent is alive.	
Message received	**Action performed**
findWebServices-Response	Obtains the list of available Web services that offer the service operation that this activity-executor agent provides. With this information, the agent fills a local list of Web services from where the agent selects the Web service that best provides the service according to the current information.
Name: CheckWebServices	**Class: CyclicBehaviour**
This behaviour is added when the agent starts its execution. Every minute, the agent executes the ping operation of the Web services contained in its local list of services. If the invocation of a Web service ping operation fails, this Web service is deleted from the list of available Web services. The agent is only registered in the DF if at least one Web service on its list of services is available. This behaviour is active while the agent is alive.	
Name: HandleRequest	**Class: CyclicBehaviour**
This behaviour is added to the task list when the agent starts its execution, and is active while the agent is alive. The agent receives and processes the following messages.	
Message received	**Action performed**
ExecuteActivity	Performs the Web service operation and traverses the process XML document, as described in Section 4.5.4. If there are more activities to perform, the agent sends ExecuteActivity messages to the activity-executor agents that provide the next activities to execute. If there are no more activities to perform, the agent sends an ExecuteActivityResponse message to the process-executor agent that initiated the service execution. If the agent is executing the last invocation of a branch of a flow activity, it sends HandleFlowSynchronization messages to all its flow partners, as described in Section 4.5.4. If the agent becomes a flow leader, it sends FlowDoneNotification messages to a set of activity-executor agents, as described in Section 4.5.4.
HandleFlow-Synchronization	Modifies the data structures that are used for performing the flow-synchronization phase. Then, the agent decides whether it is a flow-leader. If it is a flow-leader, it sends FlowDoneNotification messages to a set of flow-partner agents and continues the process execution, as described in Section 4.5.4. Otherwise, it cleans its data structures used for performing flow activities.
FlowDoneNotification	Cleans its data structures used for performing flow activities.
CancelActivity	Executes the cancellation operation of the Web service that was invoked, according to the information in the process XML document. Then, it sends the process-executor agent a CancelActivityResponse message.

5.2.7 Activity-executor agent

An activity-executor agent offers a specific atomic service. It receives ExecuteActivity messages, executes the service operation and contacts other activity-executor agents to execute the entire process. This agent uses the common, findWebServices, and executeActivity ontologies. Its actions are defined in Table 5.6.

5.2.8 UDDI agent

The UDDI agent directly accesses the UDDI service registry that is used to search for Web services that offer a service with a specific interface and namespace. It receives **findWebServices** messages from activity-executor agents, searches the UDDI registry, and returns a list of available Web services with the applicable restrictions for their invocation. This agent uses the findWebServices ontology. The actions that this agent executes are defined by the behaviours described in Table 5.7.

Table 5.7. UDDI agent behaviours

Name: UDDIResponder	**Class: CyclicBehaviour**
This behaviour is added to the task list when the agent starts its execution and is active while the agent is alive. The agent receives and processes the following message.	
Message received	**Action performed**
findWebServices	Searches the UDDI registry for Web services that offer a service with a given interface and namespace. Then, it returns a findWebServicesResponse message with a list of available Web services along with the restrictions for invocation.

5.3 Web service invocation

Activity-executor agents (AEAs) use the JAX-RPC API for Web service invocation of the Java Web Services Developer Pack (JWSDP). Web services are invoked through remote-procedure calls by using the static-stub client. Static-stub classes are created at development time from the WSDL document of the Web service represented by the AEA.

Based on the current information, an activity-executor agent selects the service to be invoked from a set of *substitutable* Web services that provide the operation it represents. Web services are substitutable if they provide the same operations through the same interface and are defined in the same namespace; thus, they are described by similar WSDL documents that differ only in the endpoint address to be invoked. The AEA uses the endpoint address of the selected Web service to create a remote-process call.

Note that activity-executor agents not only provide Web services to be integrated in a composite service, but can also offer a service whose functionality comes from legacy applications. This allows other kinds of services, not only Web services, to participate in the service composition process.

5.4 Web service registration and discovery

ASTEK uses the UDDI registry to discover available Web services to be incorporated into the composition. In order to be discovered and take part in the composition process within the ASTEK platform, Web services must be registered in a UDDI registry by following a set of rules. This section describes how Web services can be registered in a UDDI registry, as well as the process that the UDDI agent follows when searching for available Web services.

5.4.1 Web service registration

When registering Web services in a UDDI registry, providers may follow some recommendations to map WSDL documents onto the UDDI data model [14]. The use of these recommendations allows automatic service discovery.

As an example of this mapping, consider the registration of the flight booking Web service, which is an atomic service in the travel service scenario, with the restriction that it only accepts requests from executive users. The Web service is offered through the

interface defined in Figure 5-5(a), and the namespace used in the travel service scenario for the flight booking service is *http://mmarl.org/wscomposition/flightBookingService.*

(a) Definition of the interface

```
public interface FlightBookingServiceInterface extends Remote{
  public FlightBookingResponse flightBooking (
          Date minDepartureDate, Date maxDepartureDate, Date minReturnDate,
          Date maxReturnDate   , Origin origin, Destination destination,
          FlightUserInformation userInformation    )
  public String ping();
  public String cancel(String bookingNo); }
```

(b) Definition of the portType element in the WSDL document

```
<definitions ……..
targetNamespace="http://mmarl.org/wscomposition/flightBookingService"
name="FlightBookingService">
……..
<portType name="FlightBookingServiceInterface">
   <operation name="cancel" parameterOrder="String_1">
       <input message="tns:FlightBookingServiceInterface_cancel"/>
       <output message="tns:FlightBookingServiceInterface_cancelResponse"/>
   </operation>
   <operation name="flightBooking" parameterOrder="Date_1 Date_2 Date_3 Date_4
                      Origin_5 Destination_6 FlightUserInformation_7">
     <input message="tns:FlightBookingServiceInterface_flightBooking"/>
     <output message="tns:FlightBookingServiceInterface_flightBookingResponse"/>
   </operation>
   <operation name="ping">
        <input message="tns:FlightBookingServiceInterface_ping"/>
        <output message="tns:FlightBookingServiceInterface_pingResponse"/>
   </operation>
</portType>
```

(c) Definition of the service element in the WSDL document

```
<service name="FlightBookingService">
 <port name="FlightBookingServiceInterfacePort"
       binding="tns:FlightBookingServiceInterfaceBinding">
         <soap:address location="http://mmarl-11:8080/fbservice/flightBooking"/>
 </port></service>
</definitions>
```

Figure 5-5. Flight booking service definition

The service provider creates the WSDL document that defines the flight booking service by using a tool such as the JAX-RPC API from J2EE. The <portType> element of the WSDL document that defines the flight booking Web service is shown in Figure 5-5(b). Moreover, the WSDL document's <service> entry, which indicates the endpoint address where the Web service is invoked, is shown in Figure 5-5(c). If another service provider wants to offer a substitutable flight-booking Web service, it has to implement the same interface and create a WSDL document with the same portType and namespace.

A mapping of the contents of the <portType> WSDL entry to a <tModel> UDDI entry is recommended [14] to allow flexible service discovery. Figure 5-6 shows the <tModel> UDDI entry that maps the content of the <portType> WSDL entry presented in Figure 5-5(b). The <tModel> entry contains the name of the WSDL <portType> element (which is the name of the Web service interface), the location of the WSDL document that describes the Web service, and a <categoryBag> element with information grouped in <keyedReference> elements. The first <keyedReference> element contains the namespace used by the Web service, and the second element defines the type of <tModel> (portType, in this case). In <keyedReference> elements extra information must be incorporated to distinguish the Web service from the rest of the services. For example, the third <keyedReference> element in Figure 5-6 describes that the Web service is invoked when the user is an executive.

```
<tModel tModelKey="uuid:fd05b709-9bfd-05b7-b317-fa1b74b81d42" >
    <name>   FlightBookingServiceInterface  </name>
    <overviewDoc>
         <overviewURL>http://mmarl-11:8080/fbservice/flightBooking?wsdl<overviewURL>
    <overviewDoc>
    <categoryBag>
         <keyedReference   tModelKey="uuid:faf86321-0dfa-f863-f084-4dea5c81ec61"
             keyName="portType namespace"
             keyValue="http://mmarl.org/wscomposition/flightBookingService" />
         <keyedReference   tModelKey="uuid:faf888cb-effa-f888-9571-058a2f194520 "
             keyName="WSDL type"       keyValue="portType" />
         <keyedReference tModelKey="uuid:faf523ac-eaca-a252-3531-623a5a113654"
             keyName="User type"       keyValue="Executive" />
    </categoryBag>
</tModel>
```

Figure 5-6. TModel UDDI entry for the flight booking service

There is also a mapping from the contents of the <service> WSDL entry to a <businessService> UDDI entry. Figure 5-7 shows the <businessService> UDDI entry that maps the content of the <service> WSDL entry presented in Figure 5-5(b). The <businessService> entry contains the name of the Web service, the endpoint address, the tModelKey of the tModel that maps the Web service interface, and a <categoryBag> element with the type of WSDL element it maps (service, in this case), the namespace in which the Web service is defined, and the Web service name.

```
<businessService  …..
   <name> FlightBookingService </name>
   <bindingTemplates>
     <bindingTemplate
      <accessPoint URLType="http">http://mmarl-11:8080/fbservice/flightBooking</accessPoint>
         <tModelInstanceDetails>
            <tModelInstanceInfo
               tModelKey="uuid:fd05b709-9bfd-05b7-b317-fa1b74b81d42">
               <description xml:lang="en">
                   The wsdl:portType that this wsdl:port implements
               </description>
            </tModelInstanceInfo>
         </tModelInstanceDetails>
     </bindingTemplate>
  </bindingTemplates>
  <categoryBag>
    <keyedReference
         tModelKey="uuid:…. "
         keyName="WSDL type"
         keyValue="service" />
    <keyedReference
         tModelKey="uuid:…."
         keyName="service namespace"
         keyValue="http://mmarl.org/wscomposition/flightBookingService"/>
    <keyedReference
         tModelKey="uuid:…."
         keyName="service local name"
         keyValue="FlightBookingService"/>
   </categoryBag>
</businessService>
```

Figure 5-7. BusinessService UDDI entry for a provider of the flight booking service

This <businessService> entry is included in the set of services offered by the provider in the <businessEntity> entry (see Section 2.2.5).

5.4.2 Web service discovery

After receiving a **findWebServices** request message from an AEA, the UDDI agent takes the following steps to search for available Web services:

1. It invokes the *find_tModel* UDDI request to search for <portType> tModels whose name and namespace correspond to the requested Web-service interface name and namespace where the Web service is defined. The response that the UDDI agent receives is a list of tModels with their name and tModelkey.
2. For each portType tModel in the list, it invokes the *find_service* UDDI request to search for services that contain the corresponding tModelKey and whose service namespace and local service name match the requested parameters. The response that the UDDI agent receives is a list of services with the businessKey and

serviceKey identifiers, which point to the businessEntity and businessService entries that match the searching criteria.

3. It invokes the *get_tModelDetail* UDDI request with the tModelKey parameter to receive the tModel information and obtain the list of restrictions for invoking the service. This list is added to the parameters of the response message.
4. It invokes the *get_serviceDetail* UDDI request with the serviceKey parameter to receive the businessService information and obtain the endpoint address from its accessPoint value. The endpoint address is added to the parameters of the response message.
5. It invokes the *get_businessDetail* UDDI request with the businessKey parameter to receive the BusinessEntity information and obtain the name, description, and contact information. These values are added to the parameters of the response message.
6. It returns a response message to the AEA that made the request to look for available Web services. The response contains a list of elements that contains information on the Web services that match the searching criteria. Each element contains the organization name, description, and contact information of the Web service provider, as well as the service endpoint and a list of restrictions to invoke the Web service.

Note that the response message contains information on the substitutable Web services that provide a specific service operation with the same interface (portType) and namespace.

5.5 Web services implementation

Web services are implemented outside ASTEK. Although atomic Web services can be deployed in different computers regardless of the platform, for the purposes of this discussion, the services were deployed in the same computer by using the J2EE application

server. Figure 5-8 shows the J2EE deploy tool with the Web services running. These Web services were published in a local UDDI registry provided by the JWSDP.

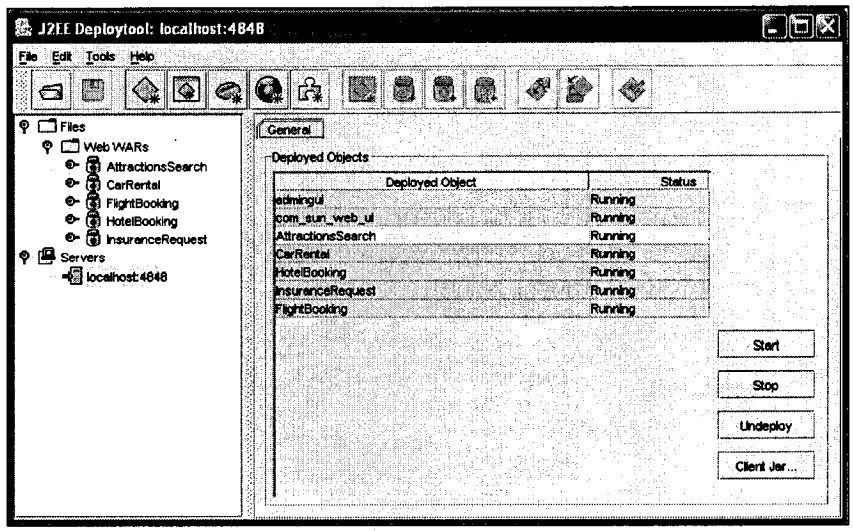


Figure 5-8. Deployment of atomic services in the travel service

5.6 Summary

This chapter presented the implementation of the agent-based Web service composition framework. ASTEK's implementation uses standard Web service technologies to create a service composition approach with high levels of scalability and flexibility. It reduces the extra work that service providers must perform to integrate their services into the service composition process.

First, this chapter presented ASTEK's architecture. Then, the implementation of software agents that integrate the framework was presented. Next, the ways in which Web services are invoked and discovered by the software agents were presented. Finally, the way in which Web services are registered and implemented by service providers was described.

Chapter 6
Performance Evaluation

6.1 Overview

This chapter presents the performance evaluation that ASTEK was subjected to. The service processing time when the same service is simultaneously executed by multiple users was selected as the parameter to evaluate from among other parameters, such as the number of messages exchanged between agents, the memory used by each agent when executing a composite service, or the scalability and flexibility levels when an activity-executor agent selects a Web service to invoke from a set of services that offer the same functionality. The service processing time represents the contributions of most factors involved in a service execution, such as network conditions and Web-service server performance.

The performance evaluation revealed that the average service processing time increases in direct proportion to the number of simultaneous users. The processing times at normal and peak workloads, 40 and 100 simultaneous users respectively, are too high. To decrease the average processing time, the number of agents that collaborate in the service execution was increased. These agents were grouped into teams in such a way that requests for service execution were assigned to different agent teams. The use of more than one team of agents deceased the service processing time.

This chapter is organized as follows. Section 6.2 describes what the performance evaluation consisted of and the scenario with which it was completed. Section 6.3 shows the service processing time when a single team of agents is used for executing services. Section 6.4 describes how the average processing time is decreased by using agent teams

and by distributing user requests among these teams. Finally, Section 6.5 summarizes the concepts presented in this chapter.

6.2 Testing scenario

ASTEK's performance evaluation reproduced the simultaneous execution of the same composite service by a varying set of users. The maximum, average, and minimum service processing times were measured for every set of simultaneous users.

The test simulates the travel service that was represented in Figure 4-2 and defined by the business process in Figure 4-4. The simulation considers that all the composite services have been invoked; thus, the itinerary is for international travel with more than three days' stay, and the user that invokes the service is a student.

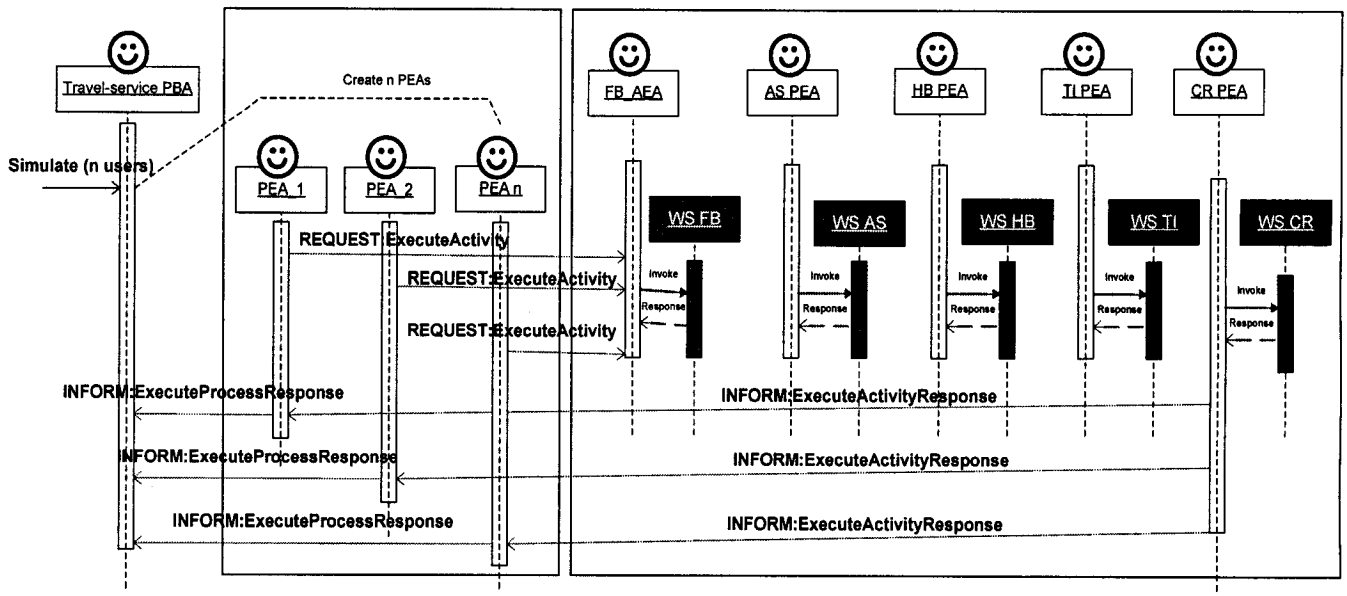


Figure 6-1. Testing scenario

Figure 6-1 depicts the testing scenario. The process-broker agent (PBA) that represents the travel service receives a request to initiate the simulation of the travel-service execution by *n* simultaneous users. For each user, the travel-service PBA creates an entry in the

processExecution table of the processes database and obtains a process ID that distinguishes the service execution from other executions. Then, for each user, the travel-service PBA creates a process-executor agent with the same user information and with a process XML document that has its corresponding process ID.

Each of the *n* process-executor agents creates a data XML document with the same inputData information, sets the *startingTime* attribute of the process XML document to the current time, and starts the business-process execution by sending *ExecuteActivity* request messages to the activity-execution agents involved in the business-process execution.

Activity-executor agents collaborate to execute *n* business processes by invoking the Web service that provides the service operation they represent, modifying the process and data XML documents with the resultant information, and sending request messages to other agents involved in the business-process execution. Eventually, one activity-executor agent reaches the end of one business process execution, sets the *endingTime* attribute of the process XML document to the current time, and calculates the *processing time*, which is the difference between the startingTime and the endingTime attributes. This processing time is set as the value of the processingTime attribute in the process XML document that represents the recently-executed business process. Finally, this activity-executor agent sends the corresponding PEA an *ExecuteActivityResponse* message that contains the resultant data and process XML documents from the service execution.

After receiving an ExecuteActivityResponse message, the process-executor agent sends an ExecuteProcessResponse message to the travel-service PBA, and terminates itself. For each of the *n* ExecuteProcessResponse messages that the travel-service PBA receives, it stores the resulting process data in the processExecution table. Thus, information from the travel-service execution by *n* simultaneous users is kept in the processExecution table.

6.2.1 Equipment conditions

The following equipment was used in the simulation:

6.2.1.1 ASTEK's computer

The computer that hosted the agent-based Web service composition framework has the following characteristics:

Processor:	AMD Athlon ™ MP 2000
Memory:	512 MB
Hard disk:	C: 28 GB, and D: 28 GB
Network card:	Realtek RTL8139 PCI 10/100 Mbps Fast Ethernet
Operating system:	Microsoft Windows 2000 Professional 5.0.2195 Service Pack 4

6.2.1.2 Web-services application server

The computer that hosted the Web services involved in the travel service has the following characteristics:

Processor:	Pentium 4, 2.40 GHz
Memory:	256 MB
Hard disk:	C: 80 GB
Network card:	SIS 900-based PCI 10/100 Mbps Fast Ethernet
Operating system:	Microsoft Windows XP Professional 5.1.2600 Service Pack 1

6.2.2 Network conditions

ASTEK was tested in a local network with the following characteristics:

Type:	IEEE 802.3 Ethernet IEEE 802.3u Fast Ethernet Ethernet 10/100 Mbps
Switching method:	Store and forward

6.3 Service processing time

This performance evaluation focused on the service processing time for a varying number of users who simultaneously invoke the same composite service. The service processing time is the number of milliseconds that the service takes to be executed.

Figure 6-2 shows the minimum, average, and maximum processing times when the travel service is executed by different numbers of simultaneous users. The maximum and average service-processing times increase with the number of simultaneous users.

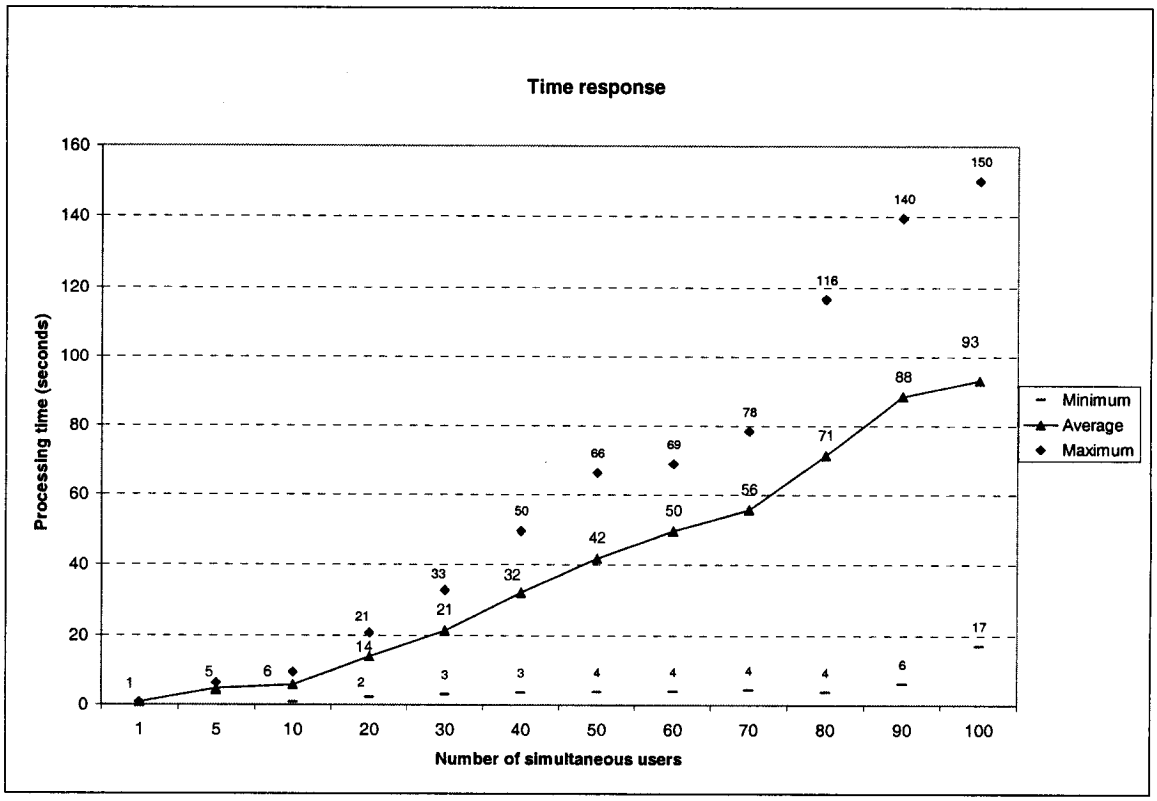


Figure 6-2. Processing time when the travel service is executed by a varying set of simultaneous users

The service processing time depends on factors such as the Web-service response time, network conditions, and the time that activity-executor agents take to carry out an invoke activity within a business process. Note that, although users simultaneously start the execution of the same service, the activity-executor agents queue activity-execution requests in their internal queue of received messages and process them one by one.

As Figure 6-2 shows, the minimum service processing times remain constant in almost all the evaluated cases. These minimum times correspond to the initial requests, which

spend less time in the activity-executor agents' queue of received messages. As the number of simultaneous users increases, activity-executor agents receive more requests and their queue of received messages also increases. Thus, some requests spend a greater proportion of their processing time waiting to be processed by the activity-executor agents.

If 40 simultaneous users is the normal workload and 100 is the peak workload, an average processing time of 32 seconds for the normal workload and 93 seconds for the peak workload could be considered unacceptable. The next section presents a method to decrease these processing times.

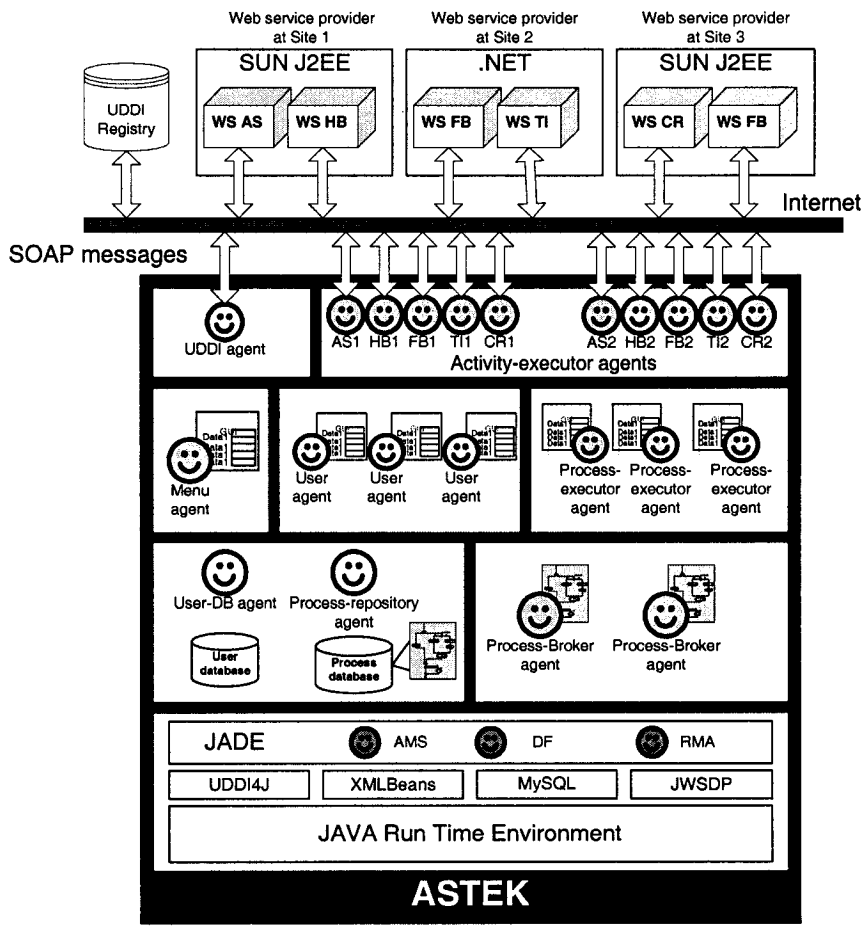


Figure 6-3. ASTEK's architecture with two teams of travel service activity-executor agents

6.4 Using teams of agents to decrease processing times

ASTEK's basic architecture was improved to reduce the processing times to more acceptable levels. Instead of using a single set of activity-executor agents to perform the service-execution requests of all the simultaneous users, multiple agents were grouped into teams and incorporated into the process. Figure 6-3 shows ASTEK's architecture with two teams of activity-executor agents that perform the composite travel service.

With the incorporation of teams of activity-executor agents (AEAs) to execute the travel service by *n* simultaneous users, the travel-service process broker agent (PBA) creates *n* process-executor agents (PEAs) and assigns them to a particular team. A PEA collaborates with only the AEAs on its team.

Figure 6-4 (below) depicts the average processing time when the travel service is executed by a different number of simultaneous users, with one, two, five and ten teams of activity-executor agents. Clearly, the average processing time decreases with an increased number of activity-executor agent teams. However, there is only a slight decrease in processing time with ten teams versus five teams of agents at both normal and peak workloads. The results also suggest that five teams of activity-executor agents optimize the processing time at normal and peak workloads by approximately 40% compared to one team of agents.

The main resource that is most likely to be exhausted with a greater number of simultaneous users and activity-executor agent teams is the memory of the computer that is hosting the ASTEK architecture. However, FIPA agent platforms can integrate agents deployed in multiple computers; therefore, the ASTEK architecture can be expanded to, and use the memory and CPU of, more than one computer.

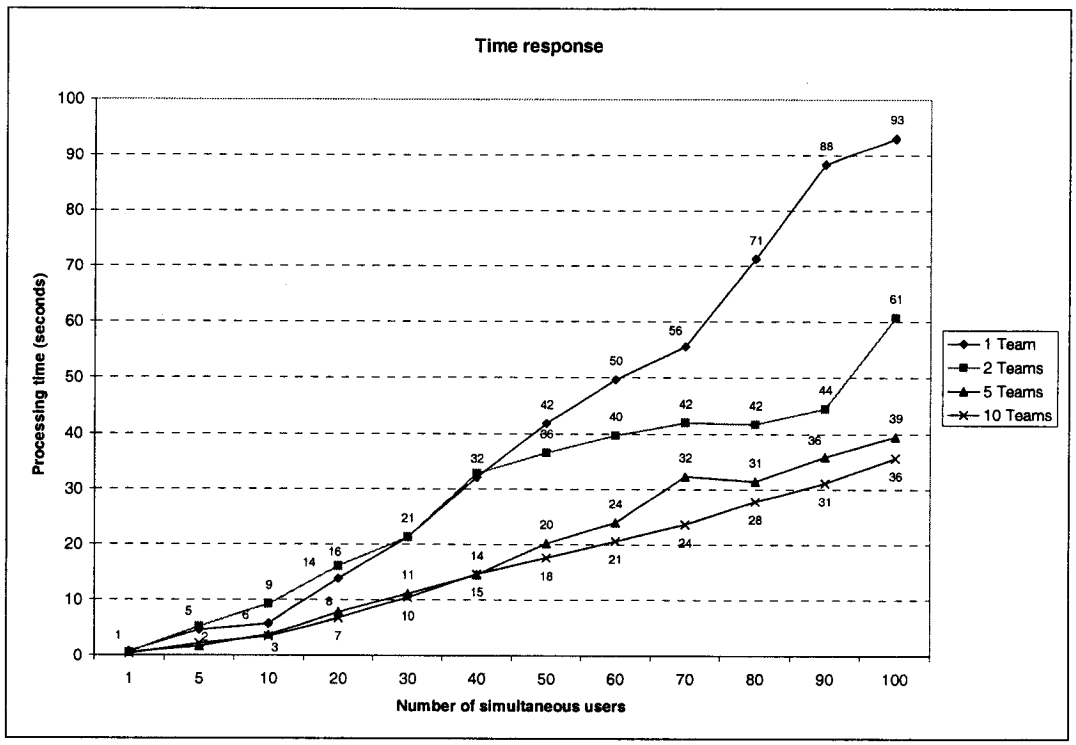


Figure 6-4. Average processing times as a function of the number of activity-executor agent teams

6.5 Summary

This chapter presented the performance evaluation that the agent-based Web service composition framework (ASTEK) was subjected to. The service processing time was the parameter that was evaluated when the framework received requests to simultaneously execute the same composite service from a varying set of users. The results indicate that the service processing time increased in direct proportion to the number of users. However, the processing times at normal and peak workloads were too high.

ASTEK's basic architecture was enhanced to decrease the average processing times by incorporating more activity-executor agents grouped into teams. Specifically, using five agent teams decreased the average processing times by 40% compared to using one agent team.

Chapter 7
Conclusions, Summary, and Future Work

7.1 Conclusions

Web service technologies enhance interconnectivity between software entities deployed in heterogeneous software architectures. These technologies are in the process of being widely accepted, and therefore widely used, by the industry. Some drawbacks must be overcome for a mature Web services architecture. Research in both academia and industry has focused on issues related to the description, composition, delivery, and discovery of Web services, as well as the security mechanism to invoke them.

Web service composition was the focus for research in this thesis. Web service composition is related to the mechanisms that promote the collaboration of individual Web services to create a software application with a functionality that is the result of integrating individual functionalities of each Web service. A composite service is a collection of atomic services, which represent service operations, arranged in such a way that their collective execution creates the complete composite-service functionality.

Software agent technology offers many advantages and conveniences to Web service composition. This thesis work utilized agent technology to propose a Web service composition framework in which the execution of a composite service is delegated to a set of software agents. Each of these agents is responsible for invoking an individual service and collaborating by exchanging the resultant data and service-execution information with their peers. The overall composite-service execution result is gathered by one of these collaborating agents.

The proactive behaviour of software agents was also used in the composite-service execution process. Collaborating agents periodically query the UDDI service registry, looking for available Web services that provide the functionality each agent represents within the framework. Then, these agents can select, from a set of Web services, the one that best fits in the service composition according to the current conditions. As a result, the framework incorporates dynamic Web service discovery into the service composition process at runtime.

The agent-based Web service composition framework (ASTEK) was subjected to a performance evaluation. Experimental results revealed that the average service-execution time was directly proportional to the number of users. However, the average processing times at normal and peak workloads were too high to be considered acceptable. The basic framework's architecture was enhanced to decrease the average processing times. The incorporation of more Web-service-invoking agents grouped into teams, and the distribution of service-execution requests across those teams, reduced the average processing times.

7.2 Contributions

This section summarizes the main contributions of this thesis.

7.2.1 Conception of an agent-based Web service composition framework

This thesis presented the design of a Web service composition framework that builds on software agent technology. Composite services are represented as business processes by using an activity diagram representation. Thus, a business-process modelling language is defined to create composite services within the framework. This language is defined in an XML Schema document. With this process modelling language it is possible to define the sequence, concurrency, and logic control that govern the execution of the atomic services to

create the entire functionality of a composite service. The atomic services—or more specifically, service operations—are modelled as actions of the activity diagram. The composite service representation contains the service operations to be executed, not the actual Web services to be invoked. For the purposes of this thesis, the terms *composite service*, *business process*, and *process* are related to the same concept and were used interchangeably.

This thesis used the travel service as an application that benefits from Web service composition. The travel service executed the flight and hotel booking, travel insurance, car rental, and attractions search atomic services. The use of this specific scenario does not cause a loss of generality; the proposed framework can be used in a variety of situations. A data representation is necessary to manipulate the partial information that results from executing atomic services, and to store the resulting information from executing the entire composite service. This data representation is specific for the context in which a composite service is in (e.g., for travel arrangements), and is defined in an XML Schema document.

The design of the proposed framework defines the software agents that are involved in the composite-service execution. These agents are the following:

1. Activity-executor agents (AEA), which represent atomic services. Every AEA is responsible for invoking a specific service operation by contacting a Web service that offers the operation the AEA represents. These agents collaborate with their peers to execute the entire business process. AEAs continuously search the UDDI service registry, through a UDDI agent, for available Web services that provide the operation they represent. An AEA is registered in the Directory Facilitator (DF) of the agent platform only if available Web services are found.

2. Process-broker agents (PBA), which represent a specific composite service. A PBA receives user requests to initiate the execution of the composite service that the PBA represents. A PBA continuously searches the DF to ensure that all the AEAs involved in executing the service that the PBA represents are available (i.e., that they are registered in the DF). PBAs register themselves in the DF only if all AEAs involved in the execution of their services are available.
3. User agents, which represent the person that wants to invoke available services. These agents present a graphical user interface (GUI) in which users can modify their personal information and select, from a list of available services, the one they want to invoke.
4. Process-executor agents (PEA), which initiate the execution of a composite service. Once a user selects an available composite service to be executed, the user agent requests the PBA that represents the selected composite service. The PBA, in turn, creates a PEA that displays a GUI in which the user can insert the information required to properly execute the service.
5. A UDDI agent, which receives requests from AEAs to find Web services registered in a UDDI registry. The searching criteria are according to the Web service's interface and namespace.

The design also defines the messages that are exchanged between software agents during execution of the composite services. These messages are defined in the ontologies that software agents use.

The main characteristics of the proposed framework are the following:

1. Services involved in the composition are organized into an executable business-process representation; however, actual invoked Web services are selected at runtime.
2. Service choreography is performed through the cooperation of software agents.
3. Service discovery is carried out by using the agent-platform Directory Facilitator to search for software agents that provide a specific service, and by using a standard UDDI service registry to search for Web services that provide a service operation through a specific interface.
4. The level of scalability is high. When a new Web service is going to be integrated into the framework, it must implement an agreed-upon service interface and be registered in a common UDDI service registry by following a set of standard recommendations. Service providers do not need to host additional modules.
5. The level of flexibility is high. A software agent that offers a specific service within the framework can select, from a set of Web services offering the same service through the same interface, the one that best fits the user requirements and the current intermediate data.
6. The composition process allows some services to fail without affecting the entire execution by incorporating optional-composite services.
7. Data and control flow representations are handled separately in two different artifacts. However, both are transmitted by using the same message.

The comparison between the Web service composition approach proposed in this thesis and other current Web service composition approaches is shown in Table 7.1 (below).

Table 7.1. Comparison between ASTEK and other Web service composition approaches

Parameter	Web Service Composition Approach				
	ASTEK	**BPEL4WS and related technologies**	**enTish**	**DAML-S**	**SELF-SERV**
Service orchestration	Reactive	Proactive	Reactive	Reactive	Reactive
Service choreography	Cooperation of PEA and AEA agents	Centralized	Cooperation of distributed entities	Centralized	Cooperation of distributed entities
Service discovery	Dynamic at runtime by using the DF and the UDDI registry	Static at development time by using UDDI registries	Dynamic at runtime by using InfoService.	Dynamic at runtime by using the DAML-S/ UDDI-based service repository	Dynamic at runtime by using the service catalogue
Level of scalability	High	Non-existent	Low	Medium	High
Level of flexibility	High	Non-existent	Medium	High	High
Service communication	Request-response messages	Request-response and one-way messages	Request-response messages	Request-response and one-way messages	Request-response and one-way messages
Recursiveness	Not supported	Supported	Not supported	Supported	Supported
Definition of interfaces	Important	Not applicable	Not applicable	Not applicable	Important
Participation of atomic services	Optional-composite service	Mandatory-composite service	Mandatory-composite service	Mandatory-composite service	Mandatory-composite service
Fault-recovery support	Provided	Provided	Provided	Not provided	Provided
Data representation	Data and control flow separated	Data and control flow together	Data and control flow together	Data and control flow together	Data and control flow together

7.2.2 Implementation of ASTEK

This thesis described the implementation of the agent-based Web service composition framework (ASTEK) and its basic architecture, including the software tools used to implement the framework as well as the databases and software agents that exist within the architecture.

Software agents are implemented by using the Java Agent DEvelopment Framework (JADE) as the agent platform. Databases are manipulated through the open source database called MySQL. XML documents are processed by using XMLBeans. Web services are invoked by using the Java Web Service Developer Pack (JWSDP) from SUN Microsystems. UDDI service registry is accessed through UDDI4J. Web services are deployed by using the J2EE 1.4 application server from SUN Microsystems.

The actions that JADE agents perform are represented as behaviours. This thesis detailed the behaviours of each software agent in ASTEK's architecture as well as the Web services' registration and discovery processes that allow their dynamic incorporation into the service composition process.

7.2.3 Composite-service execution through collaborating software agents

This thesis proposes a mechanism to execute a composite service through the collaboration of software agents. The software agents in charge of executing the service are the following: (a) a process-broker agent (PBA), which represents a composite-service within the framework; (b) a process-executor agent (PEA), which initiates the execution of an instance of a service; and (c) activity-executor agents (AEA), which represent individual services and collaborate to execute the business process in a peer-to-peer fashion.

This mechanism consists of a definition of the modelling language that will represent the composite service as a business process (in an XML Schema document), and the definition of the protocol that the AEAs follow to execute the composite service. The definition of a composite service is stored in a process XML document that follows the structural rules of the process XML Schema.

To execute a composite service, an AEA receives a request to invoke the service that it represents. Then, it invokes the Web service that best fits in the composition, modifies the data XML document with the resultant composite-service execution information, modifies the process XML document with the current information on the service execution, and analyzes the process XML document to determine the next action to execute. If the AEA detects that more atomic services must be executed, it sends request messages to the AEAs that are in charge of executing the missing atomic services. The exchanged messages contain the process and data XML documents. The execution process continues until an AEA detects that no more atomic services are needed. Then it returns to the PEA that initiated the composite service execution the resulting process and data XML documents with the final information on the composite service execution.

7.2.4 Incorporation of standard technologies for a dynamic service discovery process

ASTEK uses a UDDI service registry to dynamically locate available Web services and integrate them into the composite process. Web services are registered in the UDDI service registry by following a set of standard specifications to map WSDL to UDDI entries. A software agent, called a UDDI agent, is requested by the AEAs to find information on Web services that provide a given service through the same interface and in the same namespace. The information that the UDDI agent obtains for each matching Web service is the name, description, and contact information of the service provider, the Web service endpoint address, and a list of restrictions on invoking the service. These restrictions are used by the AESs as parameters to decide whether the Web service fits in the current composition.

7.2.5 Incorporation of optional composite services

This thesis proposed a business process representation in which atomic services are categorized as vital or negligible in the entire process execution. When an AEA detects that a vital atomic service cannot be successfully executed, it changes the execution status of the process XML document to *failed* and the process execution is cancelled. However, if the AEA detects that a negligible atomic service cannot be successfully executed, it does not change the execution status of the process and continues executing the composite service as if nothing happened.

7.2.6 Utilization of teams of software agents to decrease response times

Experimental results showed that the average time to execute a composite service is directly proportional to the number of simultaneous users invoking it. However, average times obtained at the normal and peak workloads were too high to be considered acceptable. This thesis proposed the use of teams of activity-executor agents to decrease the average composite-service processing time.

In ASTEK's basic architecture, a single set of activity-executor agents is responsible for processing all the requests to execute a composite service. An AEA queues the received requests and processes them one by one. Thus, the first requests are processed without significant delay, but the last requests spend much of their processing time waiting to be processed.

The basic architecture was improved to decrease the average processing times at normal and peak workloads. Instead of having a single set of AEAs to process all requests to execute a composite service, more AEAs grouped into teams were incorporated into the framework. Thus, the workload of service-execution requests can be distributed across the teams of activity-executor agents. Experimental results show that, with five of these teams,

the average processing times at normal and peak workloads is nearly 40% lower that with a single team.

7.3 Future work

This section lists recommendations for future work in the design and implementation of ASTEK.

1. Incorporation of asynchronous Web service calls. ASTEK considers that Web services are invoked through request-response messages in a synchronous way. An improvement could be made to enable Web services that are invoked through one-way messages, in an asynchronous manner. This would require a mechanism to match the asynchronously received messages with the corresponding process.
2. Deployment of composite services that exist in the framework as Web services. This thesis considers that composite services in ASTEK are not available outside the infrastructure. These composite services could be deployed as Web services to create broader software applications in which the atomic services are already composite services within the framework.
3. Improvement of the selection process that activity-executor agents use to determine the Web service to be incorporated into the composition. The criteria that activity-executor agents use to select, from a list of available Web services, the one to be incorporated into the composition process are parameters that are registered in the UDDI entry of each Web service. If the name of a key selection parameter is incorrectly registered in the UDDI entry, the Web service cannot be integrated into the composition. Semantics are needed if the same parameter uses different names.

Moreover, the selection process could be improved if an inference engine were used to match the Web service invocation conditions with the current execution context.

4. Incorporation of security mechanisms to invoke Web services. A standard security mechanism to invoke Web services could be integrated into the framework to enable accurate authentication and authorization.

5. Allow more user interfaces to execute composite services in the framework. In this thesis work, users are able to access the functionality that the framework offers by using a graphical user interface that a user agent provides. This user interface is only accessible when the user agent is deployed. Thus, only users that create their agent in the computer that is running ASTEK can use the framework. However, a JADE add-on could allow users to access the framework using a Web browser. Agents would be sent to user's browser as applets, allowing the user to access the framework as if he or she were on the computer that hosts the framework.

References

[1] D. KIELY, *Are Components the Future of Software?* IEEE Computer, February 1998, pp.10-11.

[2] D. S. FRANKEL, *Model Driven Architecture. Applying MDA to Enterprise Computing*, Wiley Publishing Inc., 2003.

[3] A. BROWN, S. Johnston, and K. Kelly, *Using Service-Oriented Architecture and Component-Based Development to Build Web service Applications,* Rational Software White Papers, available at: http://www-106.ibm.com/developerworks/rational/library/510.html, 2002.

[4] D.K. BARRY, *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide*, Morgan Kaufmann Publishers, 2003.

[5] W. VOGELS, *Web Services Are Not Distributed Objects*, IEEE Internet Computing, December 2003, 7(6): 59-66.

[6] H. KREGER, *Web Services Conceptual Architecture (WSCA 1.0)*, IBM Software Group, May 2001.

[7] *Web Services Architecture*, W3C Working Group Note 11 February 2004, available at: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211.

[8] F.P. COYLE, *XML, Web Services, and the Data Revolution*, Information Technology Series, ed. Addison-Wesley, 2002.

[9] *Simple Object Access Protocol (SOAP) 1.1*, W3C Note, May 2000, available at: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[10] *SOAP Version 1.2 Part 0: Primer*, W3C Recommendation, June 2003, available at: http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[11] *Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001, available at:* http://www.w3.org/TR/wsdl

[12] *UDDI Technical White Paper,uddi.org, September 2000*

[13] *UDDI Version 2.03 Data Structure Reference*, UDDI Committee Specification, July 2002, available at: http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm.

[14] *Technical Note. Using WSDL in a UDDI Registry, Version 2.0,* UDDI Specification TC, 2003, available at: http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v200-20031104.htm

[15] M. TURNER et al., *Turning Software into a Service*, IEEE Computer, October 2003, 36(10): 38-44.

[16] M. PAOLUCCI et al., *Importing the Semantic Web in UDDI*, Lecture Notes In Computer Science, Springer-Verlag Heidelberg, Volume 2512/2002, pp.225–236.

[17] *Security in a Web Services World: A Proposed Architecture and Roadmap,* A joint White Paper from IBM Corporation and Microsoft Corporation, April 2002, available at: http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/

[18] *Technical Overview of the OASIS Security Assertion Markup Language V1.1,* OASIS, March 2004, available at: http://www.oasis-open.org

[19] C. VAWTER and E. Roman, *J2EE vs. Microsoft.NET A comparison of building XML-based web services*, TheServerSide.Com, available at: http://www.theserverside.com/articles/article.tss?l=J2EE-vs-DOTNET

[20] T. AMUND, *A survey of Agent-Oriented Software Engineering.* Norwegian University of Science and Technology, May 2001.
[21] M. WOOLDRIDGE, *Agent-based software engineering.* Software Engineering. IEEE Proceedings, Feb 1997, 144(1): 26-37
[22] M. WOOLDRIDGE and N.R. Jennings, *Intelligent Agents: Theory and Practice,* The Knowledge Engineering Review, 2(10):115-152, 1995.
[23] N.R. JENNINGS and M. Wooldridge, *Agent-oriented software engineering.* Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering (MAAMAW-99), 1999.
[24] Foundation for Intelligent Physical Agents (FIPA), *FIPA Agent Management Specification,* available at: http://www.fipa.org.
[25] B. BURG, *Agents in the World of Active Web-Services,* Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Volume 2362/2002, 2002
[26] T.R. GRUBER, *A Translation Approach to Portable Ontology Specifications,* Technical Report KSL 92-71, Knowledge Systems Laboratory, Stanford University, September 1992.
[27] V. VASUDEVAN, *Comparing Agent Communication Languages,* OBJS Technical Note, Object Services and Consulting, Inc., July 1998, available at: http://www.objs.com/agility/tech-reports/9807-comparing-ACLs.html
[28] Foundation for Intelligent Physical Agents (FIPA), *FIPA ACL Message Structure Specification,* available at: http://www.fipa.org.
[29] Foundation for Intelligent Physical Agents (FIPA), *FIPA Interaction Protocol Library,* available at: http://www.fipa.org.
[30] D. A. MENASCÉ, et. al, *Scalable P2P Search,* IEEE Internet Computing, March-April 2003,83-87.
[31] D. CHAKRABORTY and A. Joshi, *Dynamic Service Composition: State-of-the-Art and Research Directions,* Technical Report TR-CS-01-19, University of Maryland, 2001, available at: http://citeseer.ist.psu.edu/chakraborty01dynamic.html
[32] C. PELTZ, *Web Services Orchestration and Choreography*, IEEE Computer, October 2003, 36(10): 46-52.
[33] S. DALAL and S. Temel, *Coordinating Business Transactions on the Web*, IEEE Internet Computing, January-February 2003, pp.30-39.
[34] G. COULOURIS et al., *Distributed Systems, Concepts and Design*, Third edition, ed. Addison-Wesley, 2001.
[35] T. ANDREWS et al., *Business Process Execution Language for Web Services Version 1.1,* IBM developerWorks, May 2003, available at: http://www-106.ibm.com/developerworks/library/ws-bpel/
[36] J.M. VIDAL et al., *Multiagent Systems with Workflows*, IEEE Internet Computing, January-February 2004, 8(1):76-82.
[37] The Business Process Management Initiative (BPMI.org), http://www.bpmi.org
[38] S. AMBROSZKIEWICZ, *enTish: An Approach to Service Composition,* Lecture Notes in Computer Science, Springer-Verlag H., Vol. 2819/2003, pp. 168-178.
[39] *Entish* Project Homepage at http://www.ipipan.waw.pl/mas/ent/
[40] S. AMBROSZKIEWICZ, *Entish: eLingua for Service Description and Composition (revised version),* April 2003, available at http://www.ipipan.waw.pl/mas/sdc-wg/.
[41] D. MARTIN et al., *DAML-S and Related Technologies,* available at http://www.daml.org/services/daml-s/0.9/survey.pdf.
[42] T. Berners-Lee, J. Hendler, and O. Lassila, *The Semantic Web,* Scientific American, vol. 248, no. 5, 2001, pp. 34-43.

[43] *Jena 2 - A Semantic Web Framework,* available at http://www.hpl.hp.com/semweb/jena.htm.
[44] M. PAOLUCCI and K. Sycara, *Autonomous Semantic Web Services,* IEEE Internet Computing, September-October 2003, 7(5):34-41.
[45] D. MARTIN et al., *DAML-S (version 0.9) Walk-Trough,* available at http://www.daml.org/services/daml-s/0.9
[46] D. MARTIN et al., *Upper Ontology for Services, Process.daml,* available at http://www.daml.org/services/daml-s/0.9
[47] B. BENATALLAH, M. Dumas, Q.Z. Sheng, Q.Z., and A.H.H. Ngu, *Declarative composition and peer-to-peer provisioning of dynamic Web services*, Proceedings of the 18th International Conference on Data Engineering (ICDE'02), March 2002, IEEE.
[48] B. BENATALLAHZ , M. Dumas, M.C. Fauvet, and H.Y. Paik, *Self-Coordinated and Self-Traced Composite Services with Dynamic Provider Selection*, School of Computer Science and Engineering, The University of New South Wales, May 2001.
[49] L. ZENG, B. Benatallah, A. Ngu, and P. Nguyen. *AgFlow: Agent-based Cross-Enterprise Workflow Management System,* In Proc. of 27th Int. Conference on Very Large Data Bases, Roma, Italy, 2001.
[50] *Unified Modeling Language: Superstructure, version 2.0,* 2nd revised submission to OMG RFP ad/00-09-02, 2003.
[51] M. MURATA, D. Lee, and M. Mani, *Taxonomy of XML Schema Languages using Formal Language Theory,* in Extreme Markup Languages, Montreal, Canada, Aug. 2001. Available at http://citeseer.ist.psu.edu/article/murata01taxonomy.html.
[52] G. CAIRE, *JADE Tutorial Application-defined content languages and ontologies,* TILab S.p.A., Jun 2002, available at: http://sharon.cselt.it/projects/jade/
[53] Java Agent DEvelopment Framework home page: http://sharon.cselt.it/projects/jade/.
[54] F. BELLIFEMINE, G. Caire, T. Trucco, and G. Rimassa, *JADE Administrator's Guide,* TILab S.p.A., December 2003, available at: http://sharon.cselt.it/projects/jade.
[55] MySQL home page: http://www.mysql.com/.
[56] XMLBeans home page: http://xml.apache.org/xmlbeans/.
[57] Java Web Services Developer Pack (Java WSDP), available at http://java.sun.com/webservices/.
[58] UDDI4J, home page: http://www.uddi4j.org
[59] F. BELLIFEMINE, G. Caire, T. Trucco, and G. Rimassa, *JADE Programmer's Guide,* TILab S.p.A., February 2003, available at: http://sharon.cselt.it/projects/jade.
[60] Foundation for Intelligent Physical Agents (FIPA), *FIPA SL Content Language Specification,* available at: http://www.fipa.org.

Appendix A

SOAP Example.

A client wants to perform a withdrawal operation in a central bank system (CBS). The client creates a request message with the user's card, the amount to be withdrawn, and the concept of the operation as parameters. The CBS returns a response message that contains the account's new balance, the transaction number, and the transaction result.

Client side.

card.setExpirationDate("**12/12/03**");
card.setHolderName("**Oscar Marquez**");
card.setNumber("**4913521487452145**");
card.setPin(**8789**);
result = requestWithdrawal(card,**52.56**,"**Phone Bill Payment**");

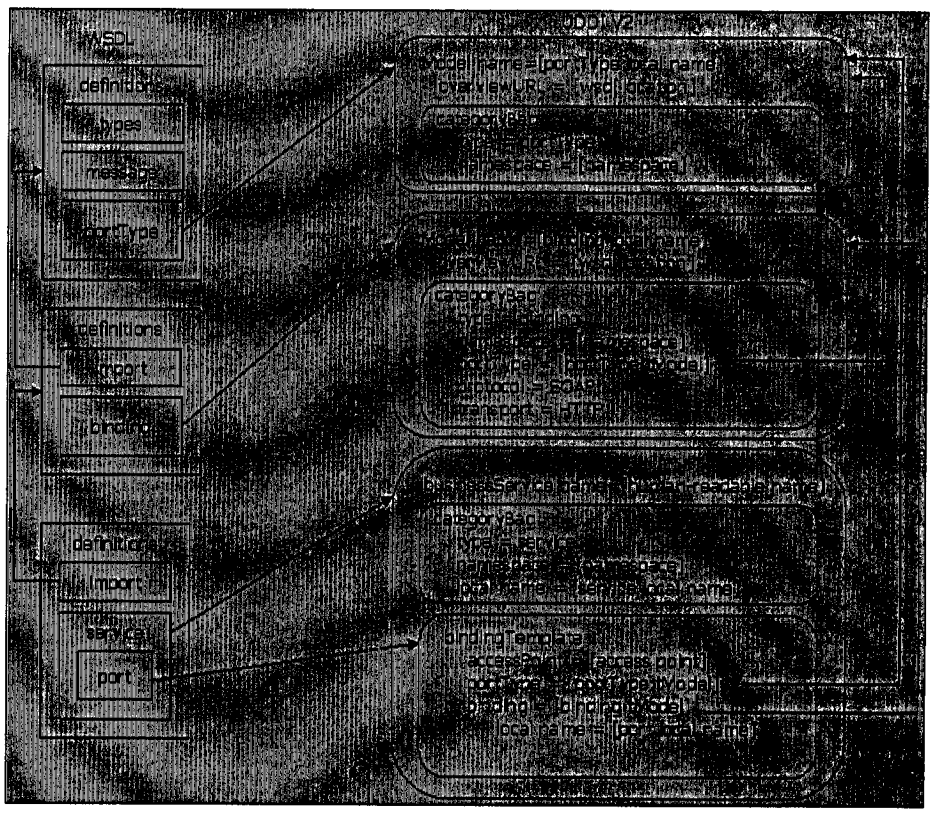
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:tns="http://mmarl.uottawa.ca/"
               xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:requestWithdrawal>
      <Card_1>
        <expirationDate xsi:type="xs:string">12/12/03</expirationDate>
        <holderName xsi:type="xs:string">Oscar Marquez</holderName>
        <number xsi:type="xs:string">4913521487452145</number>
        <pin xsi:type="xs:int">8789</pin>
      </Card_1>
      <double_2 xsi:type="xs:double">52.56</double_2>
      <String_3 xsi:type="xs:string">Phone Bill Payment</String_3>
   </tns:requestWithdrawal>
  </soap:Body>
</soap:Envelope>
```

Server side.
response.setNewBalance = **345.45**;
response.setTransactionNumber = **4567**;
response.setTransactionResult = "**WD_OK**";

```
<env:Envelope
env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:ns0="http://mmarl.uottawa.ca/">
 <env:Body>
   <ns0:Response id="ID1" xsi:type="ns0:Response">
    <newBalance xsi:type="xsd:double">345.45</newBalance>
    <transactionNumber xsi:type="xsd:int">4567</transactionNumber>
    <transactionResult xsi:type="xsd:string">WD_OK</transactionResult>
   </ns0:Response>
 </env:Body>
</env:Envelope>
```

Appendix B

WSDL-UDDI V2 data mapping [14]



Appendix C

Services Involved in the Travel Service Scenario.

The composite travel service invokes the following services:

E.1 Flight booking service

The flight booking service performs the flightBooking operation. It receives the user information and information on the planned flight. Then, it processes the request by booking the most suitable round-trip flight from the origin to the destination. The service takes into account the user information to make the flight arrangement. Finally, it returns the details of the flight booking.

E.1.1 Inputs

Invocation of the flight booking service requires the following input parameters: minimum and maximum departure and return dates, origin (country, city, and postal code), destination (country, city and postal code), user information (name, occupation, date of birth, seating preference (window, aisle or other), and favourite time to travel.

E.1.2 Outputs

Invocation of the flight booking service generates the following output parameters: booking number, airline name, flight number, seat number and position (window, aisle or other), actual departure date, actual destination arrival date, actual return date, actual origin arrival date, origin and destination airport information (airport name, address, Web page), departure gate, price, and comments.

E.1.3 Preconditions

Invocation of the flight booking service requires that the minimum and maximum departure and return date parameters are provided and congruent (e.g., no return dates before departure dates).

E.1.4 Postconditions

As a result of the invocation of the flight booking service, the actual departure date is chosen to be between the minimum and maximum departure dates. Similarly, the actual return date is chosen to be between the minimum and maximum return date. The hours of the trip are selected according to the user's favourite time to travel. The seat is selected based on the user's occupation and seating preference. The service returns the booking number, flight number and flight details. The booking number is used to cancel flight booking, if necessary.

E.2 Hotel booking

The hotel booking service performs the hotelBooking operation. It receives the destination where the user wants to stay, the arrival and departure dates as well as some user information. Then, the

service makes the hotel reservation at the hotel nearest the destination for the number of days between the arrival and departure dates. Extra services are performed based on the user occupation. The service returns a confirmation number and the details of the hotel reservation.

E.2.1 Inputs

Invocation of the hotel booking service requires the following input parameters: destination (country, city and postal code), arrival and departure dates, and user information (name, occupation and date of birth).

E.2.2 Outputs

Invocation of the hotel booking service generates the following output parameters: confirmation number, hotel name, hotel address (number, street, city, province, postal code and country), hotel Web page, hotel phone number, arrival date, departure date, number of nights' stay, price per night, total price, number of stars of the hotel, room details, extra services included in the room, extra conditions, and the distance between the hotel and destination.

E.2.3 Preconditions

Invocation of the hotel booking service requires successful execution of the flight booking service. The actual arrival and departure dates generated by the flight booking service invocation are used as parameters to invoke the hotel booking service.

E.2.4 Postconditions

The hotel booking service returns a confirmation number as well as details of the hotel booking. It locates the nearest hotel to the destination. The confirmation number is used to cancel the hotel booking, if necessary.

E.3 Travel insurance

The travel insurance service performs the *travelInsurance* operation. It executes the request by providing insurance that covers the user for travel-related problems that may arise. This service is executed only in the case of an international flight.

E.3.1 Inputs

The invocation of the travel insurance service requires the following input parameters: coverage starting and ending dates and user information (name, occupation and date of birth)

E.3.2 Outputs

Invocation of the travel insurance service generates the following output parameters: insurance number, company name, coverage starting and ending dates, emergency phone numbers, insurance conditions, insurance price, and the company's Web page.

E.3.3 Preconditions

Invocation of the travel insurance service requires successful execution of the flight booking service. The actual arrival and departure dates generated by the flight booking service invocation are used as the coverage starting and ending dates.

E.3.4 Postconditions

The travel insurance service returns the insurance number and details on the insurance. The service finds the most suitable insurance company based on the user information. The insurance number is used to cancel the insurance, if necessary.

E.4 Car rental

The car rental service performs the *carRental* operation. It makes the car reservation according to the user information. It is executed only if the user stays more than three days at the destination. The service receives the destination, the number of days that the user will stay there and the user information.

E.4.1 Inputs

Invocation of the car rental service requires the following parameters: destination (country, city, and postal code), pickup and return dates, and user information (name, occupation, and date of birth).

E.4.2 Outputs

Invocation of the car rental service generates the following output parameters: confirmation number, company phone number, company Web page, pickup and return addresses (number, street, city, province, postal code, and country), pickup and return dates, car description, insurance conditions, mileage conditions, and total price.

E.4.3 Preconditions

Invocation of the car rental service requires successful execution of flight booking service. The actual arrival and departure dates generated by the flight booking service invocation are used as the pickup and return dates, respectively.

E.4.4 Postconditions

The car rental service returns the car rental confirmation number, pickup and return addresses, pickup and return dates, and service details. The service contacts the car rental agency that has the nearest pickup and return locations to the destination. The car is selected based on the user's occupation. The confirmation number is used to cancel the reservation, if necessary.

E.5 Attractions search

The attractions search service performs the *attractionsSearch* operation. It returns a list containing attractions that may be of greatest interest to the user. It finds the attractions that are closest to the destination and that match the user's favourite types of attractions. The service also receives the arrival and departure dates in order to find the most suitable attractions at the time of the trip.

E.5.1 Inputs

Invocation of the attractions search service requires the following parameters: destination (country, city, and postal code), arrival and departure dates, and user information (name, occupation, date of birth, and preferred type of attractions).

E.5.2 Outputs

Invocation of the attractions search service generates a list of the best matched attractions. An attraction is characterized by its name, category, description, address (number, street, city, province, postal code, and country), phone number, Web page, operation hours, distance from the destination, travel times by driving and by walking from the destination, rates, and admission fees.

E.5.3 Preconditions

The attractions search service can be invoked independently of the execution of the flight booking service. It only requires accurate information about the destination and the minimum departure and maximum return dates provided by the user.

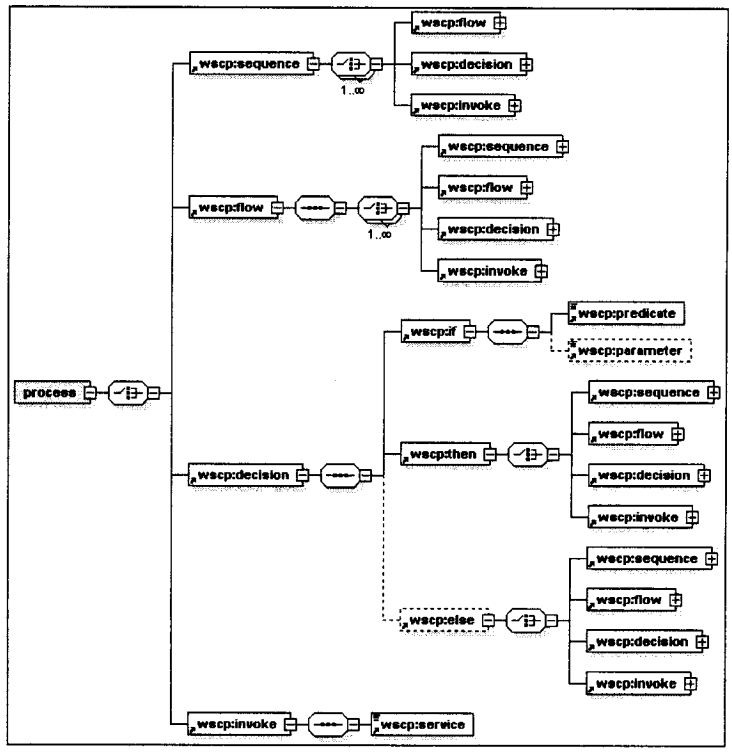
E.5.4 Postconditions

The attractions search service returns a list of attractions that the user may be most interested in visiting.

Appendix D

Business Process Description Language used in ASTEK

This appendix shows the XML Schema used to define a business process inside the ASTEK architecture (Process.xsd). A graphical representation of the XML Schema is presented before its textual representation.




```
<xs:schema targetNamespace="http://mmarl.org/XMLProcess"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:wscp="http://mmarl.org/XMLProcess">
<xs:element name="process">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="wscp:sequence"/>
      <xs:element ref="wscp:flow"/>
      <xs:element ref="wscp:decision"/>
      <xs:element ref="wscp:invoke"/>
    </xs:choice>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="status" default="inactive">
      <xs:simpleType> <xs:restriction base="xs:string">
          <xs:enumeration value="inactive"/>
          <xs:enumeration value="active"/>
          <xs:enumeration value="canceled"/>
          <xs:enumeration value="done"/>
          <xs:enumeration value="failed"/>
      </xs:restriction> </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="startingTime" type="xs:dateTime"/>
    <xs:attribute name="endingTime" type="xs:dateTime"/>
    <xs:attribute name="processingTime" type="xs:double"/>
    <xs:attribute name="failureCause" type="xs:string" use="optional"/>
  </xs:complexType> </xs:element>
<xs:element name="sequence">
```

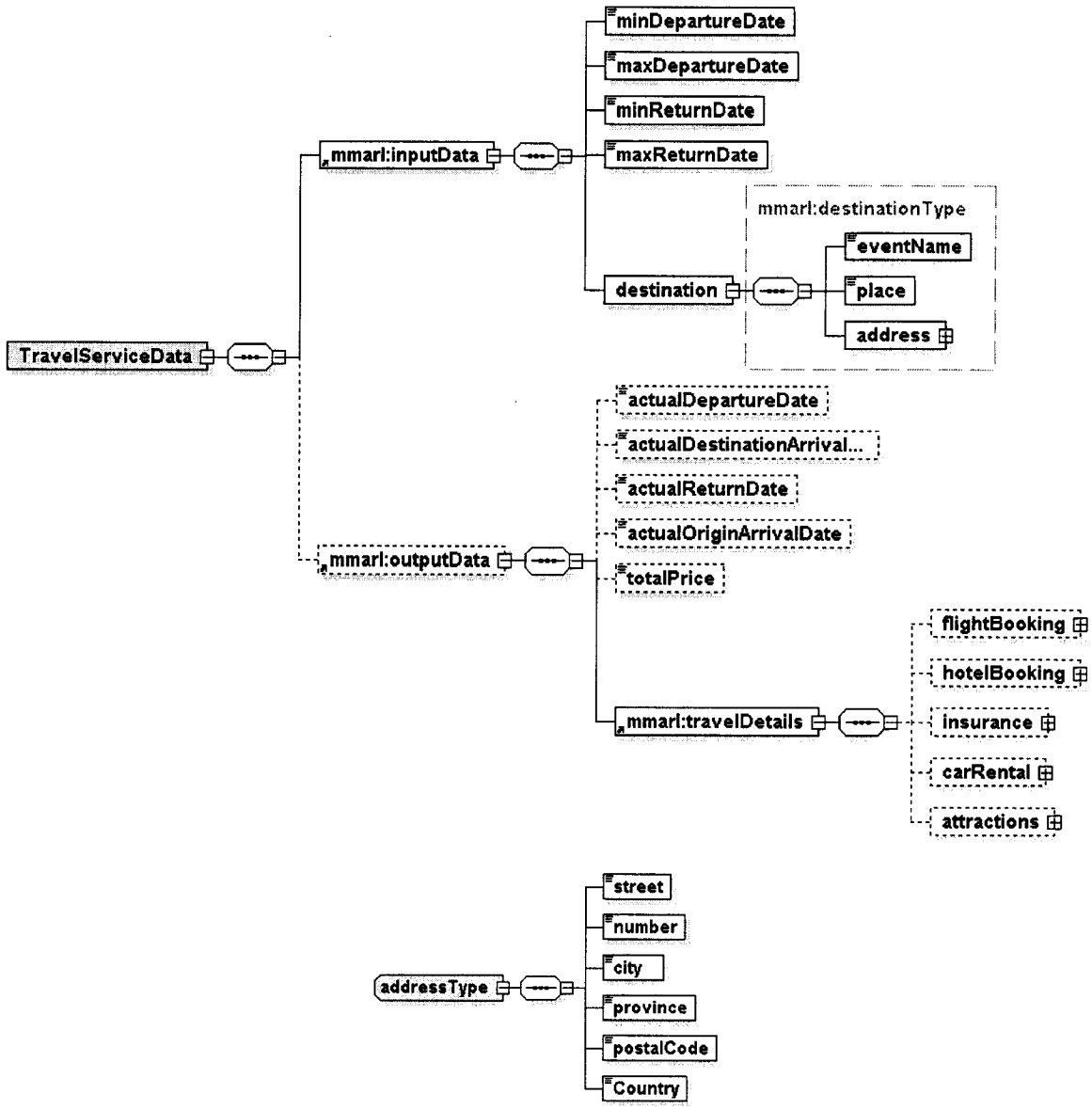
```
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="wscp:flow"/>
        <xs:element ref="wscp:decision"/>
        <xs:element ref="wscp:invoke"/>
      </xs:choice>
      <xs:attribute name="level" type="xs:ID" use="required"/>
      <xs:attribute name="status" default="inactive">
        <xs:simpleType> <xs:restriction base="xs:string">
            <xs:enumeration value="inactive"/>
            <xs:enumeration value="canceled"/>
            <xs:enumeration value="processing"/>
            <xs:enumeration value="done"/>
        </xs:restriction> </xs:simpleType>
      </xs:attribute>
    </xs:complexType> </xs:element>
  <xs:element name="flow">
    <xs:complexType>
      <xs:sequence>
        <xs:choice maxOccurs="unbounded">
          <xs:element ref="wscp:sequence"/>
          <xs:element ref="wscp:flow"/>
          <xs:element ref="wscp:decision"/>
          <xs:element ref="wscp:invoke"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="level" type="xs:ID" use="required"/>
      <xs:attribute name="status" default="inactive">
        <xs:simpleType> <xs:restriction base="xs:string">
            <xs:enumeration value="inactive"/>
            <xs:enumeration value="canceled"/>
            <xs:enumeration value="processing"/>
            <xs:enumeration value="done"/>
        </xs:restriction> </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="lastOnFinish" type="xs:string"/>
      <xs:attribute name="timeOnFinish" type="xs:dateTime"/>
    </xs:complexType> </xs:element>
  <xs:element name="decision">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="wscp:if"/>
        <xs:element ref="wscp:then"/>
        <xs:element ref="wscp:else" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="level" type="xs:ID" use="required"/>
      <xs:attribute name="status" default="inactive">
        <xs:simpleType> <xs:restriction base="xs:string">
            <xs:enumeration value="inactive"/>
            <xs:enumeration value="canceled"/>
            <xs:enumeration value="processing"/>
            <xs:enumeration value="done"/>
        </xs:restriction> </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="validation" default="undefined">
        <xs:simpleType> <xs:restriction base="xs:string">
            <xs:enumeration value="undefined"/>
            <xs:enumeration value="true"/>
            <xs:enumeration value="false"/>
        </xs:restriction> </xs:simpleType>
      </xs:attribute>
    </xs:complexType> </xs:element>
  <xs:element name="if">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="wscp:predicate"/>
        <xs:element ref="wscp:parameter" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType> </xs:element>
  <xs:element name="predicate">
    <xs:simpleType> <xs:restriction base="xs:string">
        <xs:enumeration value="ItIsAnInternationalFlight"/>
        <xs:enumeration value="StayDaysAreMoreThan"/>
    </xs:restriction> </xs:simpleType> </xs:element>
  <xs:element name="parameter" type="xs:string"/>
  <xs:element name="then">
    <xs:complexType>
```

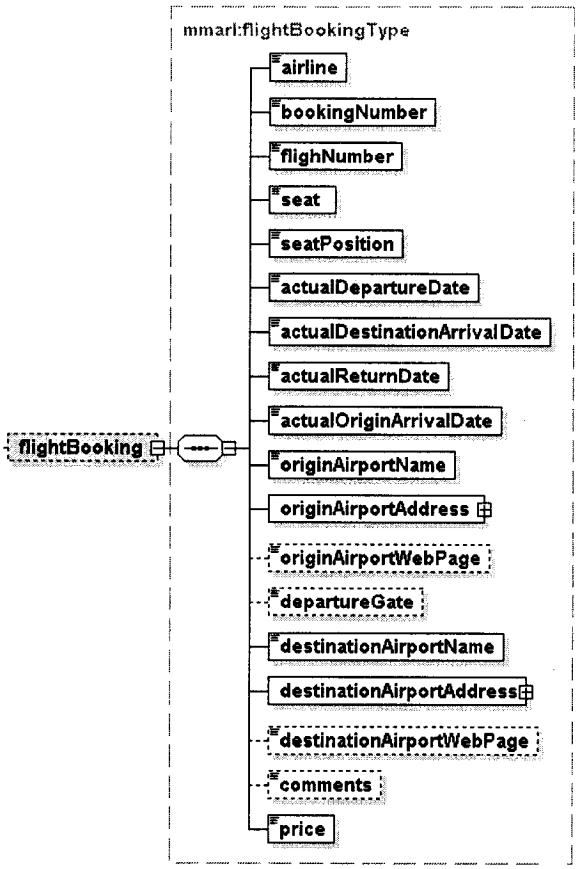
```
    <xs:choice>
      <xs:element ref="wscp:sequence"/>
      <xs:element ref="wscp:flow"/>
      <xs:element ref="wscp:decision"/>
      <xs:element ref="wscp:invoke"/>
    </xs:choice>
  </xs:complexType> </xs:element>
<xs:element name="else">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="wscp:sequence"/>
      <xs:element ref="wscp:flow"/>
      <xs:element ref="wscp:decision"/>
      <xs:element ref="wscp:invoke"/>
    </xs:choice>
  </xs:complexType> </xs:element>
<xs:element name="invoke">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="wscp:service"/>
    </xs:sequence>
    <xs:attribute name="level" type="xs:ID" use="required"/>
    <xs:attribute name="status" default="inactive">
      <xs:simpleType> <xs:restriction base="xs:string">
          <xs:enumeration value="inactive"/>
          <xs:enumeration value="canceled"/>
          <xs:enumeration value="processing"/>
          <xs:enumeration value="done"/>
          <xs:enumeration value="failed"/>
      </xs:restriction> </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="importance" default="vital">
      <xs:simpleType> <xs:restriction base="xs:string">
          <xs:enumeration value="vital"/>
          <xs:enumeration value="negligible"/>
      </xs:restriction> </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="WebServiceInvoked" type="xs:string" use="optional"/>
    <xs:attribute name="invokedBy" type="xs:string" use="optional"/>
    <xs:attribute name="invocationTime" type="xs:long" use="optional"/>
    <xs:attribute name="failureCause" type="xs:string" use="optional"/>
  </xs:complexType> </xs:element>
<xs:element name="service">
  <xs:simpleType> <xs:restriction base="xs:string">
      <xs:enumeration value="TravelInsurance"/>
      <xs:enumeration value="FlightBooking"/>
      <xs:enumeration value="HotelBooking"/>
      <xs:enumeration value="CarRental"/>
      <xs:enumeration value="AttractionsSearch"/>
  </xs:restriction> </xs:simpleType> </xs:element>
</xs:schema>
```

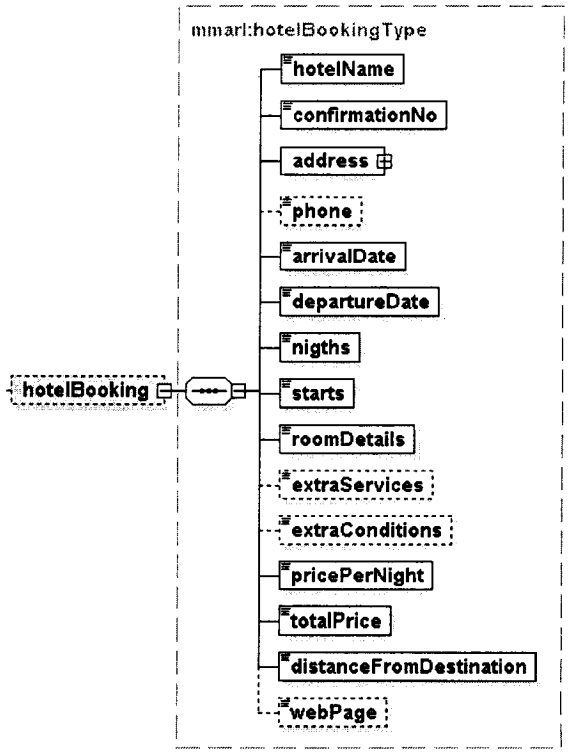
Appendix E

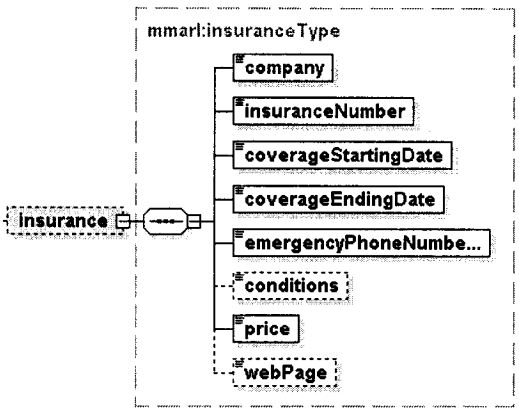
Travel Service Data Representation

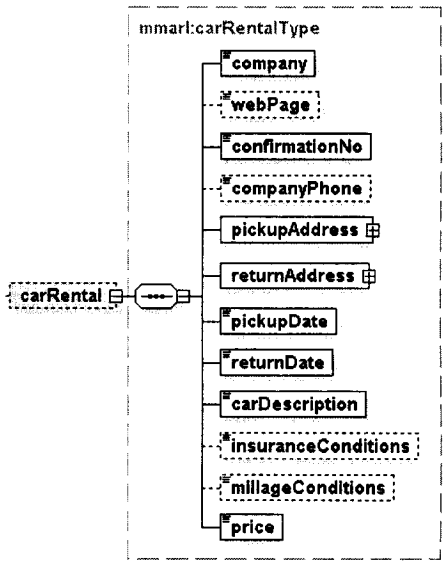
This appendix shows the data representation in the particular case of a travel service. A graphical representation of the XML Schema is presented before its textual representation.

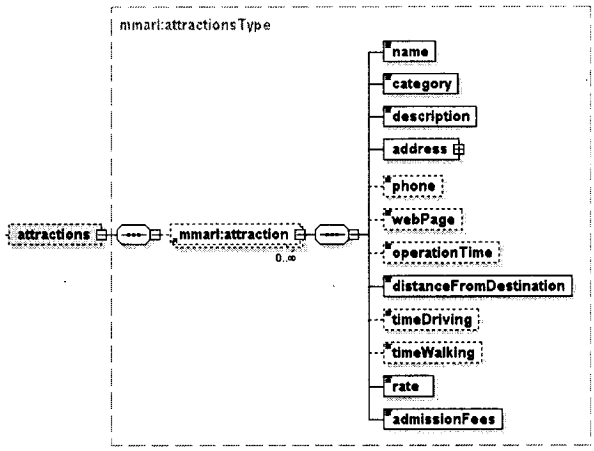



mmarl:flightBookingType
flightBooking
airline
bookingNumber
fligh Number
seat
seatPosition
actualDepartureDate
actualDestinationArrivalDate
actualReturnDate
actualOriginArrivalDate
originAirportName
originAirportAddress
originAirportWebPage
departureGate
destinationAirportName
destinationAirportAddress
destinationAirportWebPage
comments
price


mmarl:hotelBookingType
hotelBooking
hotelName
confirmationNo
address
phone
arrivalDate
departureDate
nigths
starts
roomDetails
extraServices
extraConditions
pricePerNight
totalPrice
distanceFromDestination
webPage


mmarl:insuranceType
insurance
company
insuranceNumber
coverageStartingDate
coverageEndingDate
emergencyPhoneNumbe...
conditions
price
webPage


mmarl:carRentalType
carRental
company
webPage
confirmationNo
companyPhone
pickupAddress
returnAddress
pickupDate
returnDate
carDescription
insuranceConditions
millageConditions
price




```
<xs:schema targetNamespace="http://mmarl.org/xmlTravelServiceData"
                  xmlns:xs='http://www.w3.org/2001/XMLSchema'
                xmlns:mmarl="http://mmarl.org/xmlTravelServiceData">
<xs:element name="TravelServiceData" >
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="mmarl:inputData"/>
   <xs:element ref="mmarl:outputData" minOccurs="0" nillable="true"/>
  </xs:sequence>
  <xs:attribute name="processID" type="xs:string" use="optional"/>
 </xs:complexType > </xs:element>
<xs:element name = "inputData">
 <xs:complexType>
  <xs:sequence>
   <xs:element name ="minDepartureDate" type="xs:dateTime" minOccurs="1"/>
   <xs:element name ="maxDepartureDate" type="xs:dateTime" minOccurs="1"/>
   <xs:element name ="minReturnDate" type="xs:dateTime" minOccurs="1"/>
   <xs:element name ="maxReturnDate" type="xs:dateTime" minOccurs="1"/>
   <xs:element name ="destination"     type="mmarl:destinationType"
                    minOccurs="1"/>
  </xs:sequence>  </xs:complexType > </xs:element>
<xs:element name = "outputData">
 <xs:complexType>
  <xs:sequence>
   <xs:element name ="actualDepartureDate" type="xs:dateTime"
                    minOccurs="0" nillable="true"/>
   <xs:element name ="actualDestinationArrivalDate" type="xs:dateTime"
                    minOccurs="0" nillable="true"/>
   <xs:element name ="actualReturnDate"     type="xs:dateTime"
                    minOccurs="0" nillable="true"/>
   <xs:element name ="actualOriginArrivalDate" type="xs:dateTime"
                    minOccurs="0" nillable="true"/>
   <xs:element name ="totalPrice" type="xs:double" minOccurs="0"
                                  nillable="true"/>
   <xs:element ref="mmarl:travelDetails" minOccurs="1"/>
  </xs:sequence> </xs:complexType > </xs:element>
<xs:element name = "travelDetails">
 <xs:complexType>
  <xs:sequence>
   <xs:element name = "flightBooking" type ="mmarl:flightBookingType"
                     minOccurs="0" nillable="true"/>
   <xs:element name = "hotelBooking"  type ="mmarl:hotelBookingType"
                     minOccurs="0" nillable="true"/>
   <xs:element name = "insurance"     type ="mmarl:insuranceType"
                     minOccurs="0" nillable="true"/>
   <xs:element name = "carRental"        type ="mmarl:carRentalType"
                     minOccurs="0" nillable="true"/>
   <xs:element name = "attractions"   type ="mmarl:attractionsType"
                     minOccurs="0" nillable="true"/>
  </xs:sequence>  </xs:complexType > </xs:element>
<xs:complexType name="addressType">
 <xs:sequence>
  <xs:element name = "street"      type = "xs:string"/>
  <xs:element name = "number"      type = "xs:string"/>
  <xs:element name = "city"        type = "xs:string"/>
```

```
  <xs:element name = "province"    type = "xs:string"/>
  <xs:element name = "postalCode"  type = "xs:string"/>
  <xs:element name = "Country"     type = "xs:string"/>
 </xs:sequence> </xs:complexType>
<xs:complexType name="attractionsType" >
 <xs:sequence>
              <xs:element ref ="mmarl:attraction" minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence> </xs:complexType >
<xs:element name="attraction">
 <xs:complexType >
  <xs:sequence>
   <xs:element name ="name" type="xs:string"/>
   <xs:element name ="category"        type="mmarl:categoryType"/>
   <xs:element name ="description" type="xs:string"/>
   <xs:element name ="address"     type="mmarl:addressType"/>
   <xs:element name ="phone"          type="xs:string" minOccurs="0"
                nillable="true"/>
   <xs:element name ="webPage"        type="xs:string" minOccurs="0"
                nillable="true"/>
   <xs:element name ="operationTime" type="xs:string" minOccurs="0"
                nillable="true"/>
   <xs:element name ="distanceFromDestination" type="xs:double" />
   <xs:element name ="timeDriving" type="xs:double" minOccurs="0"
                nillable="true"/>
   <xs:element name ="timeWalking" type="xs:double" minOccurs="0"
                nillable="true"/>
   <xs:element name ="rate"              type="mmarl:rateType"/>
   <xs:element name ="admissionFees" type="xs:string"/>
  </xs:sequence>  </xs:complexType> </xs:element>
<xs:simpleType name="rateType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="BAD"/>
  <xs:enumeration value="REGULAR"/>
  <xs:enumeration value="GOOD"/>
  <xs:enumeration value="EXCELENT"/>
 </xs:restriction> </xs:simpleType>
<xs:simpleType name="categoryType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="MUSEUM"/>
  <xs:enumeration value="AMUZING PARK"/>
  <xs:enumeration value="MOVIES"/>
  <xs:enumeration value="THEATER"/>
  <xs:enumeration value="BAR"/>
  <xs:enumeration value="FESTIVAL"/>
  <xs:enumeration value="OUTDOOR"/>
  <xs:enumeration value="OTHER"/>
 </xs:restriction> </xs:simpleType>
<xs:complexType name="carRentalType" >
 <xs:sequence>
  <xs:element name ="company" type="xs:string"/>
  <xs:element name ="webPage"         type="xs:string" minOccurs="0"
               nillable="true"/>
  <xs:element name ="confirmationNo"        type="xs:string"/>
  <xs:element name ="companyPhone"    type="xs:string" minOccurs="0"
               nillable="true"/>
  <xs:element name ="pickupAddress"     type="mmarl:addressType"/>
  <xs:element name ="returnAddress"     type="mmarl:addressType"/>
  <xs:element name ="pickupDate"                    type="xs:dateTime"/>
  <xs:element name ="returnDate"                    type="xs:dateTime"/>
  <xs:element name ="carDescription"        type="xs:string"/>
  <xs:element name ="insuranceConditions" type="xs:string" minOccurs="0"
               nillable="true"/>
  <xs:element name ="millageConditions"type="xs:string" minOccurs="0"
               nillable="true"/>
  <xs:element name ="price" type="xs:double"/>
 </xs:sequence> </xs:complexType >
<xs:complexType name="destinationType" >
  <xs:sequence>
   <xs:element name = "eventName"          type = "xs:string"/>
   <xs:element name = "place"              type = "xs:string"/>
   <xs:element name = "address"        type = "mmarl:addressType"/>
  </xs:sequence>  </xs:complexType>
<xs:complexType name="flightBookingType" >
 <xs:sequence>
  <xs:element name ="airline" type="xs:string"/>
  <xs:element name ="bookingNumber"         type="xs:string"/>
  <xs:element name ="flighNumber"           type="xs:string"/>
```

```
  <xs:element name ="seat" type="xs:string"/>
  <xs:element name ="seatPosition"          type="xs:string"/>
  <xs:element name ="actualDepartureDate"          type="xs:dateTime"/>
  <xs:element name ="actualDestinationArrivalDate" type="xs:dateTime"/>
  <xs:element name ="actualReturnDate"             type="xs:dateTime"/>
  <xs:element name ="actualOriginArrivalDate"      type="xs:dateTime"/>
  <xs:element name ="originAirportName"       type="xs:string"/>
  <xs:element name ="originAirportAddress"    type="mmarl:addressType"/>
  <xs:element name ="originAirportWebPage"    type="xs:string"
                  minOccurs="0" nillable="true"/>
  <xs:element name ="departureGate"   type="xs:string" minOccurs="0"
                  nillable="true"/>
  <xs:element name ="destinationAirportName"     type="xs:string"/>
  <xs:element name="destinationAirportAddress" type="mmarl:addressType"/>
  <xs:element name ="destinationAirportWebPage"    type="xs:string"
                 minOccurs="0" nillable="true"/>
  <xs:element name ="comments"       type="xs:string" minOccurs="0"
                 nillable="true"/>
  <xs:element name ="price"                        type="xs:double"/>
 </xs:sequence> </xs:complexType >
<xs:complexType name="hotelBookingType" >
 <xs:sequence>
  <xs:element name ="hotelName"            type="xs:string"/>
  <xs:element name ="confirmationNo"  type="xs:string"/>
  <xs:element name ="address"           type="mmarl:addressType"/>
  <xs:element name ="phone"   type="xs:string" minOccurs="0" nillable="true"/>
  <xs:element name ="arrivalDate"          type="xs:dateTime"/>
  <xs:element name ="departureDate"        type="xs:dateTime"/>
  <xs:element name ="nigths"                       type="xs:int"/>
  <xs:element name ="stars"                        type="xs:int"/>
  <xs:element name ="roomDetails"          type="xs:string"/>
  <xs:element name ="extraServices"   type="xs:string" minOccurs="0"
                nillable="true"/>
  <xs:element name ="extraConditions" type="xs:string" minOccurs="0"
                nillable="true"/>
  <xs:element name ="pricePerNight"        type="xs:double"/>
  <xs:element name ="totalPrice"                   type="xs:double"/>
  <xs:element name ="distanceFromDestination" type="xs:double"/>
  <xs:element name ="webPage" type="xs:string" minOccurs="0" nillable="true"/>
 </xs:sequence> </xs:complexType >
<xs:complexType name="insuranceType" >
 <xs:sequence>
  <xs:element name ="company"                  type="xs:string"/>
  <xs:element name ="insuranceNumber"       type="xs:string"/>
  <xs:element name ="coverageStartingDate"  type="xs:dateTime"/>
  <xs:element name ="coverageEndingDate"    type="xs:dateTime"/>
  <xs:element name ="emergencyPhoneNumbers" type="xs:string"/>
  <xs:element name ="conditions" type="xs:string" minOccurs="0"
                           nillable="true"/>
  <xs:element name ="price"   type="xs:double"/>
  <xs:element name ="webPage" type="xs:string" minOccurs="0" nillable="true"/>
 </xs:sequence>  </xs:complexType >
</xs:schema>
```

Appendix F

Travel-Service Confirmation E-mail

This appendix shows the content of the e-mail that is sent by the process-executor agent as a response of the service execution described in Section 5.2.6.

Composite Service Execution Confirmation

User Information
User Name: Oscar Marquez **Occupation:** Student

Service Execution Detail:

Service name: Travel Service **Process ID:** P453 **Execution Time:** 3203 ms **Final status: done**

Services that where executed:

Service Name: **AttractionsSearch**,
Level into the Process: **I11**,
Web service Invoked: **http://mmarl-11:8080/asservice/attractionsSearch**,
Importance: negligible, Invocation Time: 156ms.

Service Name: **FlightBooking**,
Level into the Process: **I12111**,
Web service Invoked: **http://travelservices:8080/fbservice/flightBooking**,
Importance: vital, Invocation Time: 203ms.

Service Name: **TravelInsurance**,
Level into the Process: **I12112**,
Web service Invoked: **http://travelservices:8080/irservice/insuranceRequest**,
Importance: vital, Invocation Time: 62ms.

Service Name: **HotelBooking**,
Level into the Process: **I122**,
Web service Invoked: **http://mmarl-11:8080/hbservice/hotelBooking**,
Importance: vital, Invocation Time: 141ms.

Service Name: **CarRental**,
Level into the Process: **I21**,
Web service Invoked: **http://mmarl-11:8080/crservice/carRental**,
Importance: negligible, Invocation Time: 1828ms.

Services that were NOT executed:
Service Name: FlightBooking. Level into the Process: I1212. Importance: vital

Travel Service Information Detail

Data document ID: P453
Input Information:
Origin:
City: Ottawa **Country:** Canada
Destination:
Event Name: MATA 2004, October 20-24

Event Place: Instituto de Computação, University of Campinas, Brazil
Address:
Street: 13084-971 Main **City:** Florianopolis **Province:** Sao Paulo
Postal Code: Caixa Postal 6176 **Country:** Brazil
Departure date:
Minimum: Sun Oct 10 09:00:00 EDT 2004 Maximum: Tue Oct 12 21:00:00 EDT 2004
Return date:
Minimum: Sat Oct 30 09:00:00 EDT 2004 Maximum: Sun Oct 31 20:00:00 EST 2004
Output Information:
Departure date: Mon Oct 11 09:00:00 EDT 2004
Destination Arrival date: Mon Oct 11 16:00:00 EDT 2004
Return date: Sun Oct 31 09:00:00 EST 2004
Origin Arrival date: Sun Oct 31 16:00:00 EST 2004
Total Price: $3925.0

Travel Details:
FLIGHT BOOKING DETAILS:
Airline: AirCanada
Flight Number: AC455 **Booking Number:** AC1501
Seat: 25. **Seat Position:** Window
Actual departure Date: Mon Oct 11 09:00:00 EDT 2004
Actual Destination Arrival Date: Mon Oct 11 16:00:00 EDT 2004
Actual Return Date: Sun Oct 31 09:00:00 EST 2004
Actual Origin Arrival Date: Sun Oct 31 16:00:00 EST 2004
Departure Gate: 20B
Origin Airport Name: Ottawa International Airport
Origin Airport Web Page: http://www.ottawa-airport.ca
Origin Airport Address:
Street: 1000 Airport Parkway Private **City:** Ottawa
Province: Ontario **Postal Code:** K1V 9B4 **Country:** Canada
Destination Airport Name: Florianopolis Airport
Destination Airport Web Page: none
Destination Airport Address:
Street: 14 RUA ARTISTA BITTENCOURT **City:** Florianopolis **Province:** SC
Postal Code: 88 020 060 **Country:** Brazil
Comments: Thanks for using our services. Enjoy our Student rates.
Price: $500.0

HOTEL BOOKING DETAILS:
Hotel Name: Florianopolis Palace Hotel
Web Page: http://reservations.webtourist.net/hotel/10007976-11254944O.html?ses=a71113aba81719c16187376cdfe4eae44234
Number of Stars: 3
Confirmation Number: HB3702
Hotel Phone: +1 888 282 1755
Arrival Date: Mon Oct 11 16:00:00 EDT 2004
Departure Date: Sun Oct 31 09:00:00 EST 2004
Hotel Address:
Street:14 Rua Artista Bittencourt **City:** Florianopolis **Province:** SC
Postal Code: 88 020 0 **Country:** Brazil
Distance from destination: 35.0 km.
Room details: With an excellent location in the heart of the Santa Catarina state capital Florianópolis, the FLOPH offers their clients comfort and personal treatment allied to the advantage of being very close to financial institutions, government headquarters such as Government Palace,

Justice Forum, Deputy's House, Private and Public Schools. Located 12 km from the Hercílio Luz International Airport and 500m from the convention center.
Extra services: Air Conditioned, Elevators, Exercise Gym, Mini Bar, In-Room Movies, Pool, Parking, Radio, Restaurant, Safe Deposit Box, Telephone, TV. Don't forget to use our Internet Services, totally free for you
Extra Conditions: No pets
Number of Nights: 19
Price per Night: 125.0
Total Price: $2375.0

INSURANCE DETAILS:
Insurance company: RBC Insurance
Web Page: http://www.rbcinsurance.com/
Insurance Number: TI5101
Coverage Starting Date: Mon Oct 11 09:00:00 EDT 2004
Coverage Ending Date: Sun Oct 31 16:00:00 EST 2004
Insurance Emergency phone numbers: 1-800-565-3129
Conditions: No smokers. You will be covered for 20 days.
Price: $100.0

CAR RENTAL DETAILS:
Company Name: Avis **Web Page:** http://www.avis.com
Confirmation Number: Avis4510
Company Phone: (55) 19-3256-2000
Pickup Date: Mon Oct 11 16:00:00 EDT 2004
Return Date: Sun Oct 31 09:00:00 EST 2004
Pickup Address:
Street: S/N Rua Bento D Arruda Camargo **City:** Campinas **Province:** Campinas
Postal Code: 13088-000 **Country:** Brazil
Return Address:
Street:
S/N Rua Bento D Arruda Camargo **City:** Campinas **Province:** Campinas
Postal Code: 13088-000 **Country:** Brazil
Car Description: Economy Car
Insurance Conditions: None
Mileage Conditions: Firts 200 Km per day are free. $0.25 per km after that.
Price: 950.0

ATTRACTIONS DETAILS:

Attraction 1 of 2:
Name: Amusing beaches
Category: OUTDOOR
Rate: EXCELENT
Description: Brazil is a country famous for its many natural wonders. Florianopolis has a special status. Geographically is it one of the only three capital cities located on an island (Santa Catarina), which is a good starting point for beach lovers.
Web Page: http://www.viagensimagens.com/engl_floripa.htm
Operation Time: N/A
Distance From destination: 2.0 km.
Time driving: 0.25 hours
Time walking: 10.0 hours.
Address:
Street: S/N N/A **City:** Florianopolis **Province:** N/A

Postal Code: N/A **Country:** Brazil
Phones: N/A
Admission fees: Free.

Attraction 2 of 2:
Name: Anthropology Museum
Category: MUSEUM
Rate: EXCELLENT
Description:
Native objects and artifacts used by natives and Azoreans immigrants.
Web Page: N/A
Operation Time: 9:00 am - 5:00 pm. Closed between Noon and 1 PM. Open on weekdays only.
Distance From destination: 2.0 km.
Time driving: 0.25 hours
Time walking: 10.0 hours.
Address:
Street:
S/N Campus Universitário – Trindade **City:** Florianopolis **Province:** NL
Postal Code: N/A **Country:** Brazil
Phone: 48-231-9325
Admission fees: Free.