Contents

Preface

Motto: *Entish - say nothing that isn't worth saying.*

Introduction and motivation of the work

Sometimes there is a tendency to oppose technology to theory and vice versa. As a matter of fact they are complementary. Technology needs theory to justify its solutions and provide sound and solid foundations, and sometimes to explain reasons for its failure. On the other hand theory needs technology to verify its ideas, theorems, and claims; if it does not, then the following citation[1] is in the right place.

As [theory] travels far from its empirical source, or still more, if it is a second and third generation only indirectly inspired by ideas coming from "reality," it is beset with very grave dangers. It becomes more and more purely anesthetizing, more and more purely l'art pour l'art*. .... In other words, at a great distance from its empirical source, or after much "abstract" inbreeding, a [theory] subject is in danger of degeneration. At the inception the style is usually classical; when it shows signs of becoming baroque, then the danger signal is up.*

We start with one of the greatest technology challenges: *How to integrate heterogeneous software applications in an open and distributed environment.* It will turn out that the crucial aspect of this challenge is, in fact, one of the most fundamental problems in the theory: *How to construct a generic open language describing data processing with precise machine understandable semantics.*

So that our primary goal is extremely ambitious, it is automatic integration of heterogeneous applications in open and distributed environment. The key requirement for a technology that could realize this goal is to make it possible to automatically discover and use an application in order to perform some given task, and automatically compose multiple applications in order to perform more complex tasks.

One may say that such technologies already exist, e.g., RPC, CORBA, and Web services to mention only the most important ones. However, none of them realizes the requirement. Although these technologies are well established as CORBA, or backed by big vendors like Web services, the dream of integrating business processes in an automatic way is still far from reality. What is the reason for that? Perhaps it is our lack of understanding of the nature of distributed systems.

It seems that this very nature is interoperability at several levels. It is not only interoperability at wire and transport levels; actually this was already realized as a working technology (e.g., Internet) quite successfully. It is also interoperability at semantic level, that is, heterogeneous applications must "understand" each other.

[1] J. von Neuman. The Mathematician. In *In the Works of the Mind.* Chicago, IL: University of Chicago, (1943)

For this very purpose a language with precise semantics that can be processed (i.e., understood) by applications is needed. This very language is the place where technology and theory meet each other.

The conclusion may be as follows. A great fundamental research is to be done both by the technology as well as by the theory. The research should be done in cooperation, that is, solution proposed by theory should be verified by technology, whereas system proposed by technology should be grounded in theory.

The work presented in the book may be seen as a small contribution to this research. A new approach to distributed systems consisting of heterogeneous components is presented. On the basis of this theoretical approach, an experimental technology for service composition is proposed. The technology is based on the new service description language with machine processable semantics. The work may be summarized in the following way.

There are two general approaches to integration of heterogeneous applications. The first one corresponds to business-to-business point of view, whereas the second one to the client's point of view. The first one is based on the assumption that applications are composed, orchestrated, or choreographed in order to create sophisticated business processes, whereas the second one assumes that applications are composed (typically on the fly) in order to realize clients' requests.

In our work the client's point of view is taken and a new experimental technology for service description and composition in open and distributed environment is proposed. The technology consists of description language called Entish, and composition protocol called entish 1.0. They are based on software agent paradigm.

The description language is the contents language of the messages that are exchanged between heterogeneous applications according to the composition protocol. The language Entish is fully declarative and has clear machine processable semantics.

A task (expressed in Entish) describes the desired static situation to be realized by the composition protocol entish 1.0. Note that a transactional semantics (similar to 3PC) is realized in the protocol.

The syntax of the description language as well as the message format are expressed in XML Schema. The language and the protocol are mere specifications. To prove that the technology does work, the prototype implementation was realized. A demo of the prototype is available as applets via web interfaces starting with http://www.ipipan.waw.pl/mas/.

The work was preceded by several papers (see (2; 3; 4; 5; 6; 7)) published in the last four years. Related work was done by SOAP + WSDL + BPEL4WS + (WS-Coordination) + (WS-Transactions), WSCI, BPML, WS-CAF, Semantic Web, DAML-S, Grid services, and Self-Serv. The references to these technologies as well as a short overview is presented in Chapter 1. The reader is cordially invited to the next chapters for details.

The work is structured in seven chapters and an Appendix where the XML sources of the enTish technology are presented.

In the first chapter, an introduction to distributed systems is presented, as well as a short overview of the most important technologies such as RPC, CORBA, software agents and multi-agent systems, Web services, and Grid services.

Chapter 2 is devoted to presentation of a new theoretical approach to distributed systems; the classical model of Client-Server is revised, and two simple running examples are introduced.

In Chapter 3, the enTish technology is presented. The description language Entish is presented formally whereas the composition protocol is introduced in an intuitive way. Two sections of this chapter are devoted to the semantics of Entish, and to the comparison of Entish to the Semantic Web.

Chapter 4 and Chapter 5 are devoted to formal and detailed presentation of implementation of the two running examples introduced in Chapter 2. These two chapters are essential to understand properly the composition protocol as well as the whole technology that is proposed.

In Chapter 6, the complete specification of the service composition protocol entish 1.0 is presented.

Chapter 7 summarizes the work by presenting an abstract architecture that may be used to implement enTish technology, as well as some details of already realized implementations.

Acknowledgments

Many people have contributed to this work by taking part in long and sometimes exiting and sometimes boring discussions with the author. The author is grateful to Leszek Rozwadowski, Tomasz Nowak, Dariusz Mikulowski, Dariusz Pawluczuk, Michal Calka, Piotr Izdebski and many others.

Special thanks go to Prof. Andrzej Barczak for his encouragement and support from Institute of Informatics of the University of Podlasie.

The author is grateful to the members of the second enTish implementation team (i.e., Dariusz Pawluczuk, Michal Calka, Piotr Izdebski, and Dariusz Mikulowski) for their implementation work and assistance.

The work was done within the framework of KBN project No. 7 T11C 040 20, as a part of the author's research in the Institute of Computer Science of the Polish Academy of Sciences, and in the Institute of Informatics of the University of Podlasie.

The author is also grateful to Steven Willmott for the cooperation within the framework of Service Description and Composition Working Group of Agentcities.NET project.

Special thanks are due to Prof. Andrzej Salwicki for motivations to writing this book.

Categories and Subject Descriptors:
C.2.4 [Computer-Communication Networks]: Distributed systems.
Keywords: Service-Oriented Architecture, service description and composition, middleware, Web services, software agents.

Chapter 1

Developing open distributed systems

1.1 Introduction to Open Distributed Systems

We focus our attention on development of open distributed systems consisting of heterogeneous components that can interoperate.

Therefore we must answer the following four basic questions:

1. What is openness (extensibility) of a distributed system?
2. What is a heterogeneous component?
3. What is interoperability between the heterogeneous components?
4. What is the goal of the distributed system to be developed?

These are some basic questions to be answered not only for our purposes but also in general in the domain of distributed systems. However, we want to constrain the analysis to the systems consisting of heterogeneous software components that communicate using a fixed transport protocol (based on TCP/IP) for sending data to each other. So that the components are networked and may be run on remote hosts on different operation systems. Since the interoperability at the level of transporting data is granted, we may focus our attention on the interoperability at higher levels that concerns automatic integration of components into a coherent system whose functionality satisfies a certain goal assumed by the system designer. The goal will be specified in the subsequent sections.

1.1.1 Openness (extensibility) of a distributed system

Openness or extensibility of the system is meant as the ability to join new components to the already running system without the need to change the existing components, so that the basic functionality of the system remains the same. The new components are supposed to interoperate with the rest of the components enriching the capability of the system.

We are interested in a system that is fault tolerant as a whole, in the sense that any component of the system may fail or be removed without causing any significant change in the basic functionality of the system. Hence, there is no single point of failure in the system. Internet and WWW are the prominent examples of such systems. It is clear that technologies for realizing such systems should be developed rather at the level of specification than as applications. It is a lesson learned from Computer Networking where interoperability is achieved by specifying standard protocols which are then implemented independently in various operation systems and interfaces.

1.1.2 Component of a distributed system

Now, it is the turn for the component of a distributed system we have in mind. Generally by component we mean a software component, e.g., object, client / server part in socket programming, software agent, and so on. Hence, we define distributed system as a collection of software components that process and exchange data. Actually, it is a very general definition. Sometimes, in the context of Internet, such distributed system is called the Cyberspace.

Let's introduce the concept of raw application. It is just an application (typically an object) we want to join to a distributed system as a component. A raw application (as an object) has several public methods that can be called. In a programming environment, the way to do it is precisely defined, and is realized by a corresponding operation system. It is clear that a raw application must be augmented with an appropriate interface (stub), if we want it to become a component of a distributed system. Hence, a distributed system is a collection of software components (i.e., raw applications augmented with appropriate stubs) that process and exchange data.

Hence, the interactions between the components are constrained to passing data. To make it possible, a communication infrastructure must be provided. For the purpose of simplicity, we abstract from protocols for transporting data (like HTTP, IIOP) and from the protocols that are in the lower layers of the network model, i.e., TCP/UDP/IP/LAN protocols, etc., We focus our attention on the interoperability problem at a higher level of abstraction. However, it is convenient to fix a data transport protocol, e.g., HTTP protocol because it is simple, ubiquitous (due to www servers) and supports most of the important data formats (i.e., data of these formats can be transported using HTTP). It is also comprehensive, that is, there are several transport methods like POST, GET, and PUT. Although our analysis will be independent of any transport protocol, it is reasonable to think in terms of these transport methods.

Since we consider components of our system as software components, we must relate our work to the Component-Oriented Programming paradigm.

1.2 Component-Oriented Programming

The idea of composing distributed systems from components, which are created and marketed independently of each other is appealing, see (21), and (22). Components are standalone objects that can plug and play across network applications, languages, tools and operating systems. The component-oriented programming (COP) paradigm defines capabilities that support a runtime environment where software components can be installed and assembled into applications in a declarative way. The COP paradigm is a natural evolution of Object-Oriented Programming (OOP) paradigm. Object-oriented programming provides only partial basis for extensibility of distributive system. Objects can be viewed as components, however some additional infrastructure is needed to enable communication between such heterogeneous remote objects.

1.3 Software agents and multi-agent systems

Software agents and multi-agent systems (MAS for short) paradigms have emerged concurrently to COP paradigm as another branch of evolution of OOP paradigm. Agent is a software object equipped with internal state consisting of several mental attitudes like Belief for expressing its knowledge, Desire for its goal, and Intentions, see the concept of BDI agent (19). Agent is proactive, i.e., depending on its state the agent is able to interact with its environment as well as with other agents. Agent's knowledge is updated after every interaction.

Multi-agent system is a system consisting of agents and environment that interact with each other. There is a special kind of interaction for realizing agent communication. The communication is in a fixed language called Agent Communication Language (ACL). Two languages are of particular interest; they are KQML (26) and FIPA ACL (27). They have incorporated several concepts from the speech acts theory (8; 20), e.g., performatives that specify message types having some intentional meaning, like request, obligation, promise, etc..

FIPA (27) is a consortium that aims at standardization of technologies related to BDI agent and MAS paradigm. The standards concern agent communication language, agent and multiagent system architecture, and agent interactions.

Usually, in a multi-agent system, the main point of interest is not agent interactions themselves but rather global system behavior emerging from long run agent interactions.

1.4 What are distributed systems for?

The natural question that must be posed is the following: What is the goal of such data processing and exchanging in a distributed system?

There are two natural answers to this question: The first one is to realize some tasks or requests, whereas the second one is to create sophisticated business processes out of the existing components that may belong to different enterprises.

Let us focus our attention on the first answer, and constrain the analysis to the systems designed to realize requests. Therefore the next question is: What or who issues such requests? Obviously, the answer is a client, that is, there must be a client application that (on behalf of a human user) wants to realize a request; it may be called client component. Requests are supposed to be realized by some other components; sometimes several components must be used. These components provide services for clients, so it is natural for them to be called service components. Hence, we are within the framework of the classic paradigm of Client Server model of distributed computing.

Let's recall that according to our definition, a distributed system is a collection of components that are raw applications augmented with appropriate stubs corresponding to a communication infrastructure. Each of such stubs represents (in the system) a raw heterogeneous applications associated with this component. That is, from the point of view of the whole distributed system, a component is visible as its stub. Let the stub of a client component be called client-agent, whereas the stub of a service component be called service-agent.

Hence, there is a client side of distributed systems as well as an opposite side, i.e., the service provider side. These two sides may influence the real world, i.e., a client by sending a request that causes data processing by the services. This very data processing may, in turn, have some effects on the real world, like money transfer, purchasing of commodities, etc..

The Client Server model is the most popular paradigm for developing distributed systems. Recently a new paradigm has emerged; it is called peer-to-peer model (P2P), see (17). It seems that it does not offer any new solution except the one that client application is integrated with server application. It means that in the same time the application may be a client to a remote server, and take the role of a server to a remote client. Although it may seem that there are no clients and no servers, i.e., there are only peers in P2P model, in fact each peer must have two different stubs, one for its client side, and one for its server side.

The most important issue of the Client Server model is the method used for client - server communication. The most common methods are based on the shared memory, message passing, and remote procedure call model.

Some distributed systems provide communication between components by modeling shared memory. Then, a global naming scheme is usually used for all shared objects in the distributed system. Read or write access to shared objects appear identical whether the object is local or remote, and takes to form of assignment. Often special hardware is used to trap assignments to or from remote objects (usually by enhancing a memory management subsystem). The shared memory paradigm is difficult for developing typical applications. Message passing or remote procedure call are more familiar and more often used.

Components may send messages to each other according to a fixed protocol. Message passing is often asynchronous. This means that one component may continue execution immediately after it has sent a message to another. It is also sometimes synchronous. Synchronous message passing involves no buffering, but it does require strong synchronization between the sender and receiver.

Remote procedure call is exactly what it sounds like. A procedure implemented in one component is made available to another component in some way, and may be called (invoked) exactly as if it were local to the caller. Remote procedure call systems follow the Client Server model very closely. A server implements the procedure. A client calls it.

Realization of communication infrastructure between heterogeneous components is called middleware. Middleware is systems software that resides between the components and the underlying operating systems, network protocol stacks, and hardware. Its primary role is to bridge the gap between heterogeneous components so that they can interoperate and may be integrated into one system.

In the past few years, several component-oriented programming middleware technologies, such as RPC, CORBA, COM/DCOM, JavaBeans/EJB, Tuplespaces, and .NET, Web services, and Grid services have emerged.

In the following sections we present short overview of RPC, CORBA, Tuplespaces, Web services, and Grid services.

1.5 The RPC Model

The RPC model describes how applications on different network nodes can communicate and coordinate activities. The paradigm of RPC is based on the concept of procedure call in a programming language. The semantics of RPC is almost identical to the semantics of the traditional procedure call. The major difference is that while a normal procedure call takes place between procedures of a single process in the same memory space on a single system, RPC takes place between a client process on one system and a server process on another system where both the client system and the server system are connected to a network.

There are several representations of the original RPC model (16). Each of the references (28), (29), (30) and (31) presents a variation on the original suitable for the exposition of their RPCs.

The basic operation of RPC may be described as follows:

1. A client application issues a normal procedure call to a client stub. The client stub receives arguments from the calling procedure and returns arguments to the calling procedure. An argument may instantiate an input parameter, an output parameter, or an input/output parameter. In this presentation of RPC model, the term input argument refers to a parameter which may be either an input parameter or an input/output parameter, and the term output argument refers to either an output parameter or an input/output parameter.

2. The client stub converts the input arguments from the local data representation to a common data representation, creates a message containing the input arguments in their common data representation, and calls the client runtime, usually an object library of routines that supports the functioning of the client stub. The client runtime transmits the message with the input arguments to the server runtime which is usually an object library that supports the functioning of the server stub. The server runtime issues a call to the server stub which takes the input arguments from the message, converts them from the common data representation to the local data representation of the server, and calls the server application which does the processing.

3. When the server application has completed the local procedure call, it returns to the server stub the results of the processing in the output arguments. The server stub converts the output arguments from the data representation of the server to the common data representation for transmission on the network and encapsulates the output arguments into a message which is passed to the server runtime. The server runtime transmits the message to the client runtime which passes the message to the client stub. Finally, the client stub extracts the arguments from the message and returns them to the calling procedure in the required local data representation.

The RPC protocol is independent of transport protocols. How a message is passed from one process to another makes no difference in RPC operations. The protocol deals only with the specification and interpretation of messages.

Time has verified this technology. Although, it was an important step in the technology development, it did not provide a flexible and ubiquitous technology for integrating heterogeneous applications in open environment.

The communication style proposed by RPC model is used also in CORBA and Web services.

1.6 CORBA

The Common Object Request Broker Architecture (CORBA), see (32), is an open distributed object computing infrastructure being standardized by the Object Management Group (OMG). CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching. CORBA ORB Architecture is composed of the following items:

- Object – This is a CORBA programming entity that consists of an identity, an interface, and an implementation, which is known as a Servant.
- Servant – This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- Client – This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object.
- Object Request Broker (ORB) – The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client's requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.
- ORB Interface – An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.
- CORBA IDL stubs and skeletons – CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.
- Dynamic Invocation Interface (DII) – This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and one-way (send-only) calls.
- Dynamic Skeleton Interface (DSI) – This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

- Object Adapter – This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB.

CORBA is a sophisticated technology providing rich functionality that includes object registry, transactions, and security to mention only a few. It is a great and verified technology for developing distributed applications within a single corporation. However, there is a problem with interoperability between CORBA components implemented by different vendors. It seems that the lesson learned from CORBA is that it is too sophisticated to become an ubiquitous technology for automatic integration of heterogeneous application in open environment.

1.7 Tuple Space

Tuple space, see (11), is a mathematical abstraction of shared memory. It can be thought of as a huge repository of data items called tuples. A tuple is a list of typed fields which can be either actual (containing a value) or formal (empty but prepared for storing a value). Together with a pool of algorithms, the tuple space makes up a so-called generative communication system. Algorithms have patterns to look for tuples they can process. If an algorithm finds a matching data item, it processes that item and puts the resulting data back into the pool as a new tuple. Since fetching and processing tuples can happen at any time, such a system is inherently parallel. ToolTalk is a communication system of Tuple space that allows to send messages into tuple space while other processes can wait for messages that fit some pattern.

It is difficult to implement a Tuple space for large systems, so it seems to be well-suited rather for specific classes of problems only. The Linda system (33), developed at Yale university, is a complete parallel programming system that is built on the tuple space paradigm. Its principles are now being used in technologies such as Jini/Javaspaces language by Sun Microsystems (34), which was modeled after the Linda concept. IBM has a similar project, called TSpaces (35).

1.8 Service Oriented Computing

Service Oriented Computing (SOC) is a quite new emerging paradigm for distributed computing that has evolved from object-oriented and component computing to enable building agile networks of collaborating business applications distributed within and across organizational boundaries. Services are autonomous platform-independent computational elements that can be described, published, discovered, orchestrated and programmed using XML artifacts for the purpose of developing distributed interoperable applications.

Service-Oriented Architecture (SOA) provides a standard programming model that allows software components, residing on any network, to be published, discovered, and invoked by each other as services. There are essentially three components of SOA: Service Provider, Service Requester (or Client), and Service Registry. The provider hosts the service and controls access to it, and is responsible for publishing a description of its service to a service registry. The requester (client) is a software component in search of a component to invoke in order to realize a request. The service registry is a central repository that facilitates service discovery by the requesters.

Web services are supposed to realize the Service-Oriented Architecture in a global networked environment. Perhaps the most popular definition of Web services can be found in IBM's tutorial (36):

Web services are self-contained, self - describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions that can be anything from simple requests to complicated business processes ... Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service.

In order to realize this vision simple and ubiquitous protocols are needed. From the service providers' point of view, if they can setup a web site they could join global community. From the client's point of view, if you can click, you could access services.

The following stack of protocols: SOAP (37), WSDL (38), and UDDI (39) is positioned to become Web services standards for invocation, description, and discovery. SOAP (in the previous versions it was the acronym for Simple Object Access Protocol) is a standard for applications to exchange XML - formatted messages over HTTP. WSDL (Web Service Description Language) describes the interface, protocol bindings and the deployment details of the service. UDDI (Universal Description, Discovery and Integration) provides a registry of businesses and web services. A UDDI service description consists of physical attributes such as name and address augmented by a collection of tModels, which describe additional features such as, for example, reference to WSDL document describing the service interface, and the classification of the service according to some taxonomies.

Since Web services are popular now, there are a lot of papers and books providing excellent introduction to the Web services; see for example (1).

Note that the Web services technology realizes service invocation in the RPC-style as well as in the document passing style.

RPC-style of Web service invocation is realized by stateless asynchronous request-response message exchange pattern. It means that a client application sends a request (as a SOAP message with WSDL document inside) to a service to invoke an operation and waits for response. The service receives the request, invokes one of its operations, and the result is returned to the client as the response. It is important that contrary to CORBA, neither the client nor the service keeps the state of the invocation session. Web services technology does not provide additional functionality as it is done in the case of CORBA. The simplicity of Web services may be seen as an advantage, however it turned out (after tree years of experience) that even that way to realize RPC is too complex to achieve interoperability. So that in the latest version of SOAP, i.e., version 1.2, the RPC-style of service invocation is considered optional. It means that there is no obligation for the vendors to implement this method; this is equivalent to the statement that the RPC-style is useless for realizing interoperability.

Hence, there is exactly one working method of service invocation in Web services technology. It is document-style method that is, in fact, extremely simple. The message exchange pattern that corresponds to this method is the following. A client sends a SOAP message to a service asynchronously, i.e., without waiting for a response. If the service wants to response, then it acts in the very similar way as the client, i.e., by sending a message to the client in the asynchronous way. Since it is one way message passing, the client as well as the service do not keep the state of the conversation session.

Let us summarize the current state of the Web service technology.

- SOAP provides a universal message format and document style service invocation method.
- WSDL is an interface definition language without possibility to describe what a service does, i.e., there are no means in WSDL to express (in a generic way) the type of operation the service performs.
- Since operation types cannot be expressed in a generic way, UDDI (as service registry) provides only partial description and classification of Web services.

The conclusion is that service discovery must be done by a user. Once the user discovers an appropriate service, the service can be invoked automatically just by sending a message, however coordination, transaction, and exception handling must be realized by the client application associated to the user.

From the Web services technology point of view, the functioning of a service is extremely simple: Wait for messages; once you get a message you may send a response asynchronously. The client is obliged to implement additional functionality, or there must be special service interfaces (acting as clients towards services) that implement such functionality. This was realized by the following technologies: BPEL4WS, WS-Coordination, WS-Transaction, WSCI, BPML, and WS-CAF that will be discussed in the next Section.

The main drawback of the Web services technology is that the discovery process can not be realized automatically. Whereas the key requirement for service integration is to make it possible to automatically discover and use services in order to perform some given task, and automatically compose multiple services in order to perform more complex tasks. The basic stack of Web services does not realize this requirement.

1.9 Web service composition

A brief overview of the most important technologies for service composition, based on SOAP+WSDL+UDDI stack, are presented below.

1.9.1 BPEL4WS

BPEL4WS (BPEL for short), see (40), is a process modeling language designed to enable a service composer (a programmer) to aggregate Web services into an execution. There are abstract and executable processes. Abstract processes are useful for describing business protocols, while executable processes may be compiled into invokeable services. Aggregated services are modeled as directed graphs where the nodes are services and the edges represent a dependency link from one service to another. Canonical programmatic constructs like SWITCH, WHILE and PICK allow to direct an execution path through the graph. BPEL was released along with two other specs: WS-Coordination and WS-Transaction. WS-Coordination describes how services can make use of predefined coordination contexts to subscribe to a particular role in a collaborative activity. WS-Transaction provides a framework for incorporating transactional semantics into coordinated activities. WS-Transaction uses WS-Coordination to extend BPEL to provide a context for transactional agreements between services. A composite service realized in BPEL can itself be exposed as a service, i.e., has its own WSDL interface. Now, BPEL is revised by OASIS as WSBPEL (43).

1.9.2 DAML-S

DARPA project DAML-S, see (44), is a DAML+OIL ontology for describing Web services. It aims at making Web services computer-interpretable, i.e., described with sufficient information to enable automated Web service discovery, invocation, composition and execution monitoring. The DAML-S Ontology comprises ServiceProfile, ServiceModel, and ServiceGrounding. ServiceProfile is like the yellow page entry for a service. It relates and builds upon the type of content in UDDI, describing properties of a service necessary for automatic discovery, such as what the services offers, and its inputs, outputs, and its side-effects (preconditions and effects). ServiceModel describes the service's process model, i.e., the control flow and data-flow involved in using the service. It relates to BPEL and is designed to enable automated composition and execution of services. ServiceGrounding connects the process model description to communication- level protocols and message descriptions in WSDL.

1.9.3 Web Services Choreography Interface (WSCI)

Web Services Choreography Interface (WSCI), see (45), is an XML-based language for Web services collaboration. It defines the overall choreography describing the messages between Web services that participate in a collaborative exchange. WSCI describes only the observable interactions between Web services. A single WSCI document only describes one partner's participation in a message exchange. Usually, a WSCI choreography includes a collection of WSCI documents, one for each partner in the interaction. It is interesting that there is no single controlling process managing the interaction. WSCI could be viewed as a layer on the top of WSDL. Each action represents a unit of work, which typically would map to a specific WSDL operation. Hence, WSCI would describe the interactions among these WSDL operations. The W3C has recently announced a Web service choreography working group that considers WSCI. Currently, WSCI is under revision by Web Services Choreography Working Group of W3C, see (46).

1.9.4 Business Process Management Language (BPML)

The Business Process Management Language (BPML), see (47), is a meta-language for describing business processes. The BPML specification can be loosely compared to BPEL, providing similar process flow constructs and activities. Basic activities for sending, receiving, and invoking services are available, along with structured activities that handle conditional choices, sequential and parallel activities, joins, and looping. BPML incorporated the WSCI protocol. WSCI could be used to describe the public interactions whereas the private implementations could be developed with BPML. Both BPML and WSCI share the same underlying process execution model and similar syntaxes.

1.9.5 Self-Serv

Self-Serv, see (9), is an academic based framework for declarative Web service composition using statecharts, where resulting composite services can be executed in a decentralized way. The concept of service communities is used to form alliances among a potentially large number of services performing the same operation types. The underlying execution model allows services participating in a composition, to collaborate in a peer-to-peer fashion in order to assure that the control and data flow dependencies of the composition schema (expressed by a statechart) are respected.

1.9.6 XSRL

XSRL - Web Services Request Language, see (49), is a formal language for expressing requests against e-marketplace registered Web services. The language is an amalgamation of the Internet XML query language and AI planning constructs. XSRL includes constructs such as alternative activities, vital vs. optional activities, preconditions, postconditions, invariants, and expression operators over quantitative values that can be employed at the user-level when formulating a goal. This approach realizes the automatic generation, verification and execution of plans in order to achieve the goals. It is built on the top of BPEL. It is also an academic based approach.

1.9.7 WS-CAF

Web Services Composite Application Framework (WS-CAF), see (48), is a collection of three specifications – Web Service Context (WS-CTX), Web Service Coordination Framework (WS-CF), and Web Service Transaction Management (WS-TXM) designed to solve problems that arise when multiple Web services are used in combination(composite applications) to support information sharing and transaction processing.

Web Service Context (WS-CTX) provides an open, common, interoperable runtime mechanism to manage, share, and access context information among related Web services.

Web Service Coordination Framework (WS-CF) defines a software agent to handle context management. Web services in a composite application register with a coordinator to ensure messages and results are correctly communicated and allow, e.g. the success or failure of an individual service to be tied to the success or failure of the larger unit of work comprising multiple Web services.

Web Service Transaction Management (WS-TXM) defines three distinct transaction protocols that can be plugged into the coordination framework for interoperability across existing transaction managers, long running compensations, and asynchronous business process flows. It also includes an innovative solution to bridge different transaction models (e.g. MQ Series, JMS).

Summing up this Section, the key requirement for service integration is to make it possible to automatically discover and use services in order to perform some given task, and automatically compose multiple services in order to perform more complex tasks. None of the technologies presented above realizes this reqiurement.

There is another quite new attempt to realize this requirement; it is Open Grid Service Architecture presented briefly below. Although it is also based on the basic protocol stack of Web services, the original roots of this technology are in Grid computing, i.e., in integrating distributed and heterogeneous computing resources. This new initiative is considered as an alternative to the composition technologies presented in the previous section as well as an enhancement of the basic protocol stack of the Web services. Since the Grid services are relatively new, it it worth to devote some space an time to present some details of this new technology.

1.10 Open Grid Service Architecture

Open Grid Service Architecture (OGSA) is an open standard managed by the GGF - The Global Grid Forum standards body, see (50).

The objective of OGSA, see (52) is to provide a common infrastructure for managing resources across distributed and heterogeneous platform by defining open published standard interfaces and protocols. The foundation of OGSA is Web services. OGSA is build on the model of service-oriented architecture (SOA).

The overview of OGSA/OGSI is based on an article (51) from IBM developer-Works.

1.10.1 The OGSA architecture

Four main layers comprise the OGSA architecture. Starting from the bottom:

- Resources – physical resources and logical resources
- Web services, plus the OGSI extensions that define grid services
- OGSA architected services
- Grid applications

Physical and logical resources layer. The concept of resources is central to OGSA and to grid computing in general. Resources comprise the capabilities of the grid, and are not limited to processors. Physical resources include servers, storage, and network. Above the physical resources are logical resources. They provide additional function by virtualizing and aggregating the resources in the physical layer. General purpose middleware such as file systems, database managers, directories, and workflow managers provide these abstract services on top of the physical grid.

Web services layer. The second layer in the OGSA architecture is Web services. Here's an important feature of OGSA: All grid resources – both logical and physical – are modeled as services. The Open Grid Services Infrastructure (OGSI) specification defines grid services and builds on top of standard Web services technology. OGSI exploits the mechanisms of Web services like XML and WSDL to specify standard interfaces, behaviors, and interaction for all grid resources. OGSI extends the definition of Web services to provide capabilities for dynamic, stateful, and manageable Web services that are required to model the resources of the grid.

OGSA architected grid services layer. The Web services layer, with its OGSI extensions, provide a base infrastructure for the next layer – architected grid services. The Global Grid Forum is currently working to define many of these architected grid services in areas like program execution, data services, and core services.

Grid applications layer. Over time, as a rich set of grid-architected services continues to be developed, new grid applications that use one or more grid architected services will appear. These applications comprise the fourth main layer of the OGSA architecture.

1.10.2 Extending Web services for grid

Let's look more closely at the two main logical components of OGSA – the Web services-plus-OGSI layer, and the OGSA architected services layer. Why are they separated like this? The GGF OGSA working group believed it was necessary to augment core Web services functionality to address grid services requirements. OGSI extends Web services by introducing interfaces and conventions in two main areas.

- First, there's the dynamic and potentially transient nature of services in a grid. In a grid, particular service instances may come and go as work is dispatched, as resources are configured and provisioned, and as system state changes. Therefore, grid services need interfaces to manage their creation, destruction, and life cycle management.

- Second, there's state. Grid services can have attributes and data associated with them. This is similar in concept to the traditional structure of objects in object-oriented programming. Objects have behavior and data. Likewise, Web services needed to be extended to support state data associated with grid services.

1.10.3 OGSI components

Open Grid Services Infrastructure (OGSI) introduces an interaction model for grid services, and provides a uniform way for software developers to model and interact with grid services by providing interfaces for discovery, life cycle, state management, creation and destruction, event notification, and reference management. Whether a software developer is developing a grid service or an application, the OGSI programming model provides a consistent way for grid software to interact.

Let's take a closer look at the interfaces and conventions OGSI introduces.

Grid services that implement this interface provide a way to create new grid services. Factories may create temporary instances of limited function, such as a scheduler creating a service to represent the execution of a particular job, or they may create longer-lived services such as a local replica of a frequently used data set. Not all grid services are created dynamically. For example, some might be created as the result of an instance of a physical resource in the grid such as a processor, storage, or network device.

Because grid services may be transient, grid service instances are created with a specified lifetime. The lifetime of any particular service instance can be negotiated and extended as required by components that are dependent on or manage that service. The life cycle mechanism was architected to prevent grid services from consuming resources indefinitely without requiring a large scale distributed "garbage collection" scavenger.

Grid services can have state. OGSI specifies a framework for representing this state called Service Data and a mechanism for inspecting or modifying that state named Find/SetServiceData. Further, OGSI requires a minimal amount of state in Service Data Elements that every grid service must support, and requires that all services implement the Find/SetServiceData portType.

Service groups are collections of grid services that are indexed, using Service Data, for some particular purpose. For example, they might be used to collect all the services that represent the resources in a particular cluster-node within the grid.

The state information (Service Data) that is modeled for grid services changes as the system runs. Many interactions between grid services require dynamic monitoring of changing state. Notification applies a traditional publish/subscribe paradigm to this monitoring. Grid services support an interface (NotificationSource) to permit other grid services (NotificationSink) to subscribe to changes.

When factories are used to create a new instance of a grid service, the factory returns the identity of the newly instantiated service. This identity is composed of two parts, a Grid Service Handle (GSH) and a Grid Service Reference (GSR). A GSH is guaranteed to reference the grid service indefinitely, while a GSR can change within the grid services lifetime. The HandleMap interface provides a way to obtain a GSR given a GSH.

To summarize this section, the OGSA architecture enhances Web services to accommodate requirements of the grid. It would be interesting to compare Grid services to CORBA. Although Grid services is not a complete technology, it seems to be more complex than CORBA, i.e., it provides much richer functionality than CORBA. It may be seen as an advantage, however the main reason for CORBA failure to be a ubiquitous technology for integrating heterogeneous applications was its very rich functionality.

The key requirement for service integration is still to make it possible to automatically discover and use services in order to perform some given task, and automatically compose multiple services in order to perform more complex tasks. Although the OGSA/OGSI augments the basic Web services stack with a lot of useful functionality, the requirement is still not satisfied.

1.11 Chapter summary

This completes a short survey of the most important paradigms and technologies concerning distributed systems. As we have seen above there are many paradigms and technologies for developing open distributed systems. However, none of them is the right technology for integrating heterogeneous applications in open environment. It seems that in order to find the reason for that and analyze these existing solutions, a common methodological framework is needed. A proposal of such framework is introduced in the next Chapter.

Chapter 2

A New Approach to Developing Open Distributed Systems

One may ask why yet another approach is needed? Although, it is too early to sentence the Web services to failure, some conclusions may be drawn from more than three years of the history of Web services.

- The RPC-style of service invocation turned out to be an obstacle to achieve interoperability.
- WSDL does not describe generic types of operations performed by services. Although this is a truism, it must be explicitly stated. Therefore an automatic service discovery is not possible.
- Since the proposed service architecture is extremely simple, all the hard work of realizing service integration is put either on the client's back (by using BPEL), and / or on additional interfaces (WS-Coordination, WS-Transactions, WSCI, WS-CAF) managed by a third party.

Perhaps these are sufficient reasons for revising the existing methodology starting with its very roots, that is, with the Client Server model of distributed computing.

2.1 Client Server model revised

We are going to revise the general model of Client Server in the spirit of Service-Oriented Architecture. The revision is done by specifing the basic parts of the Client Server model in a generic way.

Like all models of distributed systems, the Client Server model consists of software components and infrastructure (middleware) for providing communication and interoperability between the components. However, in the case of Client Server model, there are two basic kinds of components, namely, client components, and service components. A client sends a request to a sever hosting some services. The request is passed to one of the services, which may realize the client's request by invoking its operation. Generally, there are two basic methods for service invocation. The first one is the RPC-style method based on synchronous request-response, whereas the second one is just one-way asynchronous document passing.

The essence of these two methods is that data and some control instruction are sent as a single request to a service via server. The problem arises if the service cannot or do not want to invoke its operation. In CORBA this problem was solved by providing additional functionality, whereas in Web services (especially in the document passing style) there is no standard mechanism to cope with such problems. It would be polite and obviously much more efficient for the client to ask a service if it can realize the client's request before sending data and control instructions.

2.1.1 Request revised

Hence, it seems reasonable to revise the notion of request as well as service invocation method. This may be done in the following manner.

1. A client sends a query to a service asking the service if it can produce output data satisfying some conditions.

2. If the service is able and agrees (that is, it commits) to do so, it sends to the client a response specifying the input data needed to produce the output specified in the query.
3. Then, if the client's data do satisfy the input specification, the data may be sent to the service by the client.
4. Finally, the service may realize its commitment, i.e., process the data according to the client's intention expressed in the query.

It is important to note that there are two phases of this method of service invocation. The first phase (item 1 and 2) consists in arranging an invocation of service operation, whereas the second phase (item 3 and 4) is the execution of the service operation according to the arrangement done in the first phase.

The problem is how to realize that, i.e., What are the key requirements? It seems that it is a generic language for expressing queries (i.e., clients' intentions), and services' commitments as responses to the clients' queries.

2.1.2 Client and service revised

Once we have revised the notion of request, let us also revise the notion of service and client. There is an asymmetry between client and service. Service processes data whereas client realizes requests by arranging data processing done by services.

Usually, a service component consists of two parts:

- A raw application that processes data.
- A generic stub that represents the application in the infrastructure.

The raw applications may be heterogeneous and remote, that is, created by different programmers on different programming platforms and operation systems. These applications may have been running on remote hosts. The generic stubs must be standardized and be common, i.e., implemented according to one standard, so that they must be considered as a part of the infrastructure. The task of a raw application is simple; it takes the input data and produces output data. The task of the stub is to arrange the input data to be delivered to the raw application, as well as to deliver the output data to its destination. This arrangement (called control flow) needs not be integrated with data passing (called data flow) as it is done in Web services.

A client component consists also of two parts:

- A raw application that may represent a user, or be just another application that wants to realize a request.
- A generic stub that represents the raw application in the infrastructure.

The task of the client's raw application is to issue the request and prepare the initial data needed (if there are any) to realize the request. The task of the stub is to find out an appropriate service (or services) and arrange its (or their) invocation in order to realize the request.

A service is represented in the infrastructure by its stub that is called service-agent, whereas a client is represented in the infrastructure by its stub that is called client-agent. A client application may send a request to its agent. The client-agent is responsible for realizing the request by communicating with the service-agents. In this way, services are used, and requests are realized. This is the very functionality that the infrastructure is supposed to perform. Since the agents must communicate and understand each other, this is the very reason for them to be standardized.

Note that one and the same raw application may be represented by a service-agent and a client-agent, i.e., it may have two stubs of different kinds, so that from one stub point of view it may be visible as a service whereas from the second one as a client.

2.1.3 Middleware

Middleware is a term used sometimes to denote the infrastructure needed to provide interoperability between heterogeneous applications. We adopt this term. In our case the middleware consists of client-agents, service-agents, communication infrastructure, and broker. Having such rough model in mind let's go into details, i.e., to the generic components of the infrastructure.

Two kinds of components have already been mentioned, namely, client-agents, and service-agents. The next components are communication infrastructure and a mechanism for realizing clients requests. The mechanism may be called a broker. Broker is a conversation protocol between agents. It is a specification of message / data exchange between parties that participate in conversation as well as specification of state change of the message sender and recipient. Hence, client-agent as well as service-agent may have its internal state of the conversation session. Since this state is a part of the conversation protocol, its format must be defined explicitly, standardized, and implemented in the agents.

It is important to notice that control flow (realized by message exchange) may be separated from data flow. These two flows may be implemented by using different methods, e.g., control flow by the method POST, whereas the data flow by the method GET of the HTTP protocol.

The message format, the message contents language, the message transport, and the data transport together constitute the communication infrastructure.

Usually, the broker (as a conversation protocol) consists of the following generic phases:

1. Publication phase - service-agent publishes the type of the operation performed by the associated raw application.

2. Discovery phase - client-agent discovers a service performing operation of a given type.

3. Coordination phase - client-agent arranges service-agents to coordinate invocations of their associated applications.

4. Execution phase - the coordinated invocations are realized. This phase may implement transactional semantics.

In order to perform the publication and discovery phase, an auxiliary component is needed, i.e., a service registry where a service-agent can publish its operation type, and a client-agent can discover services needed to realize its request. This component should perform matching between client-agent queries and service operation types. Let this component be called *matching-agent*.

2.1.4 Description language

It seems that the crucial component of the middleware is the message contents language. This very language constrains and sometimes determines the format of the message and agent's state as well as the conversation protocol. It is the very language in which clients' requests are to be expressed, and service operation types (i.e., what a service does perform) are described. In this very language the states of agents should also be described. Although request definition language, and service description language may be considered as separate languages, it is clear that there must be a strong semantic interrelation between them, i.e., between requests and operation types, because requests are supposed to be realized by performing appropriate operations. This very semantic interrelation must be implemented in some way as a translation. In order to avoid this, it is natural to have one description language (DL for short).

It is important what the scope of the description is, i.e., what is described and how it is described in DL. It is clear that DL should describe the so called Cyberspace, i.e., the world where data are processed and exchanged by raw applications represented by the agents. Client's request describes a class of situations of that world, that is, the client's desire is to be in one of these situations. Since client-agents and service-agents are parts of the world, their states may also be described in DL. Therefore it seems that middleware technology is determined by DL, i.e., by how detailed description of the world can be expressed in DL. To grasp this claim, consider the following simple examples of requests. Since we have not introduced any syntax of DL these examples are presented informally.

- The first example: The distributed system in question is the WWW. A client component (a browser) sends a request message to a www server:
  ```
  GET /somedir/page.html HTTP/1.1
  Host:  www.someschool.edu
  ...
  ```
  In fact, the HTTP protocol specifies the language for expressing requests as well as for describing services. These very services are www servers that provide files upon request. The description language is extremely simple and is based on the concept of URL. Roughly speaking, an URL consists of the name of a host and a path to a file.

- The second example: *"Call the method x of an object of class Y with parameters Z".* This kind of requests may be realized by RPC-style technology, e.g., CORBA, RMI, or the basic web services stack, i.e., SOAP+WSDL+UDDI. In the case of Web services, the description language is WSDL. The language describes service interfaces (methods and its parameters) and binding details, i.e., how to communicate with a service.

- The third example: *"Book a flight from Warsaw to San Francisco; the departure date is to be between December 26 and December 31, 2003".* Usually, to realize this kind of requests, the RPC model is not sufficient. The client's request specifies a large set of possible situations, so that there is a large number of possible computations (in RPC-style, or document passing style) that lead to these situations. However, to perform one of such computation, the client must create an order (as the input parameter) so that he must know and specify, in advance, the airline name, the flight number, the price, the exact departure date, etc., . What is even more interesting here is the fact that this kind of request can not be expressed by a language that describes only service interfaces (signatures), like IDL of CORBA, and WSDL of Web services.

2.1.5 Some conclusions

In Web services technology, a protocol for invoking services consists of two kinds of messages, that is, request and response. However, in general case, service invocation protocol may consist of much more message kinds.

There are two crucial aspects of the invocation protocol: Data flow and control flow. In the case of Web services these two flows are integrated in one message exchange pattern realized synchronously in RPC-style, i.e., client sends a request to a service and waits synchronously for a response. Data and corresponding control are included in one request/response message. Since the request/response pattern is simple and synchronous, the client as well as the service are stateless, i.e., they do not store the current state of the message exchange session.

However, this pattern is not obligatory. It is possible to have a message oriented asynchronous invocation, where data flow and control flow are separated as well as two different transport protocols are used; one for transporting messages containing control data (processed by agents), and the second transport protocol for passing data that are processed by raw applications. Client Server model is general so that it does not impose any constrains on the service invocation protocol and the underling communication infrastructure except the very one that services are used and clients' requests are realized.

The revision of the Client Server model presented above may be seen as abstractions of the most popular RPC-style of service invocation. The first abstraction concerns control flow, i.e., the scope of the language for expressing control can be much more rich, i.e., it needs not be so narrow as it is in the case of IDL of CORBA, and WSDL of Web services.

The second abstraction is that there may be a separation between control flow and data flow as well as asynchronous message and data passing.

The third abstraction concerns the concept of request and the fixed message exchange pattern of the RPC-style and document passing style. It needs not be the simple request-response pattern where the parties that exchange messages are stateless. According to this abstraction the message exchange pattern may be called a conversation protocol. So that agents participating in a conversation may have states that are relevant to the conversation, i.e., passing data or a message of a certain kind causes state change of the sender/recipient.

The final conclusion of this section is that the key aspect of the middleware is a description language that determines the conversation protocol.

2.2 Agents and implementation of the middleware

We have introduced three kinds of agents that participate in the conversation protocol of the middleware, namely, service-agent, client-agent, and matching-agent. The goal of the protocol is to realize requests issued by clients. The agents are stubs of (client, service, and registry) applications that communicate with each other by sending messages according to the conversation protocol using the underlining communication infrastructure. So that an agent represents its raw application in the middleware, and must implement its part of the conversation protocol. Hence, the whole conversation protocol is implemented in the agents. If specification of a conversation protocol and a communication infrastructure is complete and agents implementations fully satisfy the specifications, then the agents can interoperate, i.e., they talk to each other according to the protocol so that services are used and clients' requests are realized.

Since the middleware consists of agents that can be implemented independently, it seems that the revised Client Server model proposes a general methodology for developing open, heterogeneous and distributed systems with no single point of failure like Internet and WWW.

We are going to introduce an experimental technology for realizing such middleware, i.e., description language, message and state format, and a conversation protocol. In the next section, simple working examples are presented to explain the idea of the middleware.

2.3 Running examples

Although the running examples are simple and natural, it is not easy to develop open and extensible systems that could realize such kinds of examples. After presenting the examples we will discuss how existing technologies like BPML4WS, WSCI, DAML-S, as well as many others, could be applied in this case.

The first example presents a scenario of booking a flight by a client.

2.3.1 Example 1

There was a client called C1. Its intention was to book a flight from Warsaw to San Francisco; the departure was scheduled on Dec. 31, 2003. The client wanted to arrange its request by Dec. 15, 2003. The client expressed the request in a description language. Since we have not introduced any syntax of DL, let the request be presented informally as the following formula:
phi =
"invoice for ticket (flight from Warsaw to San Francisco, departure is Dec. 31, 2003) is delivered to C1 by Dec. 15, 2003"

Then, the request formula (i.e., *phi*) was delegated to a the client-agent, say C1-agent. The request became the goal of the C1-agent. The C1-agent set the request formula as its first intention, and was looking for a service that could realize it. First of all, the C1-agent sent the query: *"C1-agent's intention is phi"* to a matching-agent called infoService. Suppose that infoService replied that there was a travel office called FirstClass that could realize C1-agent's intention. Then,

the agent sent again the formula *"C1-agent's intention is phi"* however, this time to the agent representing the FirstClass; let the agent be called FC-agent. Suppose that FC-agent replied with the following commitment:
"FC-agent commits to realize phi,
if (order is delivered to FC-agent by Dec. 15, 2003 and
the order specifies the flight (i.e., from Warsaw to San Francisco, departure Dec. 31, 2003) and
one of the following additional specification of the order is satisfied:
(airline is Lufthansa and the price is 300 euro)
or
(airline is Swissair and the price is 330 euro)
or
(airline is LOT and the price is 280 euro))"

Let *psi* denote, the formula after *"if"* inside (...) parentheses. The formula *psi* is the precondition of the commitment. Once the C1-agent had received the info about the commitment, the C1-agent considered the intention *phi* as arranged to be realized by FC-agent, and then the C1-agent put the formula *psi* as its current intention, and looked for a service that could realize it. Let us notice that the order specified in the formula *psi* could be created only by the client via its C1, that is, the client had to decide which airline and price should be chosen, and the complete order was supposed to include details of a credit card of the client. Hence, the C1-agent sent the following message to C1: *"C1-agent's intention is psi"*.

Suppose that C1 replied to the agent: *"C1 commits to realize psi, if true "* Note that now the client C1 acted as a service, that produces data of the form of orders. The C1-agent considered the intention *psi* as arranged to be realized by C1. Since the precondition of the C1 commitment was the formula *"true"*, a workflow for realizing C1-agent's request was already constructed. Once C1 had created the order and sent it to FC-agent, the FC-agent passed the order to its raw application, i.e., to the FirstClass office. FirstClass produced an appropriate invoice and returned it to the FC-agent. The FC-agent sent it to C1.

It was supposed (in the protocol) that once a service-agent had realized a commitment, it sent the confirmation to the C1-agent. Once the C1-agent had received all confirmations (i.e., from C1 and FC-agent), it got to know that the workflow was executed successfully. In order to complete this distributed transaction, the C1-agent sent synchronously the final confirmation to all the services engaged in the workflow. This completes the first example.

A complete description of the implementation of this example is presented in Chapter 4.

2.3.2 Example 2

For the purpose of presentation the example is a bit simplified. A description of the implementation of this example is presented in Chapter 5.

The main difference between the first and the second example is that in Example 1 there is only one service (called FirstClass), whereas in this example there are two services (called AnyBook, and BigBank) that are composed into workflow. So that the purpose of this example is to show how services can be composed (on the fly) into a workflow in order to realize a request.

Client C1 was going to purchase a book entitled "The Lord of the Rings". It wanted to arrange the request by Dec. 15, 2003. Client C1 expressed the request in a formal description language; suppose that it was the following formula:
phi =
"invoice for the book "The Lord of the Rings" is delivered to C1 by Dec. 15, 2003"

Then, the request formula (i.e., *phi*) was delegated to C1-agent. The request became the goal of the C1-agent. The C1-agent set the request formula as its first intention, and was looking for a service that could realize it. The C1-agent sent the following query to an matching-agent called infoService:
"C1-agent's intention is phi"

Suppose that infoService replied that there was a bookstore called AnyBook that could realize C1-agent's intention. Then, the agent sent again he formula: *"C1-agent's intention is phi"* however, this time to the service-agent representing AnyBook that was called AB-agent. Suppose that AB-agent replied with the following commitment:
"AB-agent commits to realize phi,
if (
(order is delivered to AB-agent by Dec. 14, 2003
and
the order specifies the title as "The Lord of the Rings" and the price as 70 euro)
and
(payment confirmation of 70 euro for the book is delivered to AB-agent by Dec. 14, 2003)
)"

Let *psi* denote, the formula after *"if"* inside (...) parentheses. The formula *psi* is the precondition of the commitment. Once the C1-agent had received the info about the commitment, the C1-agent considered the intention *phi* as already arranged to be realized by AnyBook. Then, the C1-agent decomposed the formula *psi* into the following two subformulas:
psi1 =
"(order is delivered to AB-agent by Dec. 14, 2003
and the order specifies the title as 'The Lord of the Rings' and the price as 70 euro)"

psi2 =
"(payment confirmation of 70 euro for the book is delivered to AB-agent by Dec. 14, 2003)"

These subformulas were set as the current intentions of the C1-agent. Then, the C1-agent was looking for services that could realize them. The order specified in the formula *psi1* could be created only by the client via its service-agent C1, that is, the client had to specify the destination address for the book delivery. Hence, the C1-agent sent the following message to the service-agent C1:
"C1-agent's intention is psi1"

Suppose that the service-agent C1 replied to the agent:
"C1 commits to realize psi1, if true"

The C1-agent considered the intention *psi1* as already arranged to be realized by the service-agent C1. There was no precondition of the commitment.

Then, the C1-agent was looking for a service that can realize the second intention, i.e., *psi2*. Suppose that C1-agent got to know from the infoService that the BB-agent (representing service BigBank) can realize its intention. So that the C1-agent sent to the BB-agent the following message:
"C1-agent's intention is psi2"

Suppose that the BB-agent replied:
"(BB-agent commits to realize psi2,
if
(payment order for 70 euro for the book is delivered to BB-agent by Dec. 14, 2003)
)"
Then, the formula:
psi21 =
(payment order for 70 euro for the book is delivered to BB-agent by Dec. 14, 2003)
became the next intention of C1-agent.

The C1-agent sent the following message to the service-agent C1:
"C1-agent's intention is psi21"
Suppose that the service-agent C1 replied with the following commitment:
"C1 commits to realize psi21, if true"
Hence, a workflow for realizing C1-agent's request had already been constructed and was ready to be executed.

Then, C1 (as service-agent) created the order and sent it to AB-agent, and created the payment order and sent it to BB-agent. BigBank produced the appropriate payment confirmation and sent it via BB-agent and AB-agent to the service AnyBook. AnyBook produced the invoice and sent it via AB-agent to C1.

C1-agent received confirmation from the service-agent of C1, from BB-agent, from AB-agent, and once again from the service-agent C1, and then sent synchronously the final confirmation to these service-agents to complete the distributed transaction. This completes the second example.

2.4 How to implement the examples?

It seems that the scenarios presented in the examples are quite natural and ubiquitous, that is, they may be considered as an attempt to automate the usual way a human user would follow in order to realize such kinds of requests. So that it seems reasonable to develop a technology for realization of such requests. In order to do so several problems must be solved.

The first problem is the request form. Actually, a client's request (from our examples) consists of two parts. The first part is the initial query, like *Book a flight ...* or *Purchase a book by ...* . A query, to be expressed in description language DL, specifies only the result (the final data) to be received by the client. This part corresponds to the first phase of the conversation protocol consisting of discovery appropriate service-agents and sending queries to them.

Usually, specification of the query is only partial, and does not determine the final data. The query must be sent (as a client's intention) to a service-agent associated with a raw application that may produce a data specified in the query. The application must be intelligent (i.e., to understand the query) to be able to answer the query. If the application can produce the desired output data, then the application returns a commitment, i.e., the constrains on the input data that must be delivered to the application in order to produce the output data specified in the query. The constrains on the input data are decomposed into next client's intentions in the form of queries, so the client-agent is looking for another services that can answer these queries; and so on. This query process is propagated back until all new queries reach the client. Then, the client obtains the specifications of the initial data that must be created in order to realize its request. Again the specifications are expressed in description language.

Note that simultaneously to the query back propagation process, the service-agents (that participate in this process) form a workflow for realizing the client's request. It means that service-agents coordinate data flow and processing that will happen if the client creates the initial data satisfying the specifications, and sends them to the next service-agents in the workflow.

These very initial data constitute the second part of the client's request and correspond to the second phase of the conversation protocol where the workflow (arranged in the previous phase) is executed.

We propose the following requirements for a technology realizing such simple scenarios as well as more sophisticated ones:

- Since the queries and answers are passed between applications, the description language for expressing queries and answers must be generic and independent of data types and types of operations performed by applications.

- Since any raw application must answer a query concerning the output it can produce, it must be augmented with additional functionality for processing such queries and producing answers to them. Hence, the description language must have precise, machine processable semantics.

It is clear that according to the requirements, service invocation must consist of two phases: In the first phase a service processes a query, produces an answer, and gets ready to process input data specified in the answer to the query. In the second phase the input data are sent to the service, are processed, and the output data are passed to the place specified in the query. Although the order of the phases is fixed, they may be performed asynchronously as well as the steps of each phase may be performed asynchronously.

The service invocation described above is consistent with the revised Client Server model. However, it seems that it is completely different from the invocation method proposed by the RPC model and document passing style. Does it mean that the technologies that are based on RPC-style and document passing style cannot realize the scenarios presented in the examples? Of course, they can, however the possible realizations are enormously complex. The reason is the lack of generic description language capable of expressing queries and answers. This means that in order to implement the conversation protocol, the types of data and types of operations that are specific for these very examples must be hardcoded in the implementation. Hence, it is impossible to implement the protocol in a generic way, if there is no generic description language.

Let us explain how the examples can be implemented using technologies, e.g., BPEL4WS and WSCI, that are based on the Web services. The service architecture corresponding to the standards SOAP+WSDL consists of a collection of operations (methods) that can be called. Although there may be semantic and functional interrelations between these operations, it cannot be expressed in WSDL where only operation signatures can be specified. Usually, an operation, that for example, processes orders and produces appropriate invoices, is accompanied by an operation that processes the queries concerning the capabilities of the original operation. Although, these two operations belong to the same service there is no way to express their semantic interrelation. Moreover, the query / answer format is hard-coded in implementation of the associated operation. Hence, there is no generic way to describe data types as well as the operation types. It is not possible to specify (in WSDL) the abstract functions implemented by the operations, i.e., what the operations do. (Note, there was an attempt to introduce it to WSDL see (53).) Service provider itself must classify these operations according to the taxonomies available in UDDI.

An alternative approach is to enrich service architecture and service description, and go beyond the traditional RPC-style and simple document passing, i.e., to allow more sophisticated protocols for service invocation. However, to do so a generic service description language is needed.

The conclusion is that the running examples can be implemented by technologies that are based on the SOAP+WSDL+UDDI protocol stack, however the implementation can not be universal, i.e., it must be dedicated to specific data and operation types occurring in the examples. The reason is that in order to do so in a generic way there must be a universal language for data and operation description. WSDL describes only signatures of operations performed by services. Client must express a query in a language that must be understood by heterogeneous services whereas a service must express an answer that must be understood by a heterogeneous client. Hence, there must be a common universal agent conversation language describing data processing.

If the running examples are regarded as simple and natural so that it is expected that more sophisticated scenarios should be realized in an universal and automatic way, then it is clear that something more is needed than the basic Web services protocol stack. It is not a critique of SOAP+WSDL as a technology realizing RPC-style and document passing style of service invocation. In fact, it is a perfect standard for this kind of service invocation. However, in order to realize the idea of Web services in its full dimension, that is, especially the aspect of automatic service integration, a more sophisticated service invocation protocol is necessary.

2.5 Chapter summary

The final conclusion of this section is that there is a need to go beyond the classic RPC-style and simple document passing. The Web services protocol stack and the integration technologies that are based on this stack are insufficient to realize automatic integration of heterogeneous applications in open and distributed environment.

The purpose of the revision of the Client Server model was to abstract from the existing service invocation methods in order to obtain a methodological basis to propose new methods.

The final conclusion of this chapter may be as follows. A generic language is needed for describing data processing and exchange by heterogeneous applications in open and distributed environment. The language must have precise semantics that can be processed automatically.

Chapter 3

enTish: A New Approach to Service Description and Composition

3.1 Beyond the RPC-style and simple document passing

It would be useless to revise the Client Server model without proposing a technology different from technologies based on the RPC-style and simple document passing. So that we are going to propose a new service invocation protocol that was roughly described in the running examples presented in Chapter 2, Section 2.3.1 and 2.3.2.

Sometimes service invocation protocol is called Message Exchange Pattern, see (54). In the technologies that follow the RPC-style of service invocation, the invocation protocol is based on the synchronous request-response pattern, and in the case of Web services the protocol is stateless.

We propose a service invocation protocol that is much more sophisticated. The protocol consists of two phases: The first one is called *query phase* whereas the second one is called *execution phase*. The query phase consists of the following two steps:

1. The client sends a query to the service specifying the desired output.
2. Then, the service answers with the specification of the input required to produce the desired output.

The execution phase consists of the following three steps:

1. The client creates input data according to the specification in the service's answer and sends to the service the pointer to this input data, e.g., the URL where the input data are stored.
2. The service downloads the input data and produces output data.
3. Then, the service sends to a client (or another service) the pointer where the output data are stored.

Note that service invocation method is a part of the conversation protocol that is described in detail in Chapter 6.

There are four basic parts of a technology implementing the revised Client Server model: Description language DL, agent's state format, message format, and conversation protocol. The fifth part is transport protocol. Although the technology should be independent of the transport protocol, it is convenient to fix one as an example. Of course, HTTP is the protocol of the first choice.

Before we specify formally these four parts, let us fix our attention on the raw application of the service component for a while. First of all it is convenient to consider service component that performs exactly one operation. In our running examples a raw application is intelligent in a sense. That is, it can answer queries concerning the operation it performs. This functionality must be implemented in this very component. Since the service-agent is supposed to be a generic stub independent of the raw application, this functionality must be implemented as a part of the raw application of the service. This is the crucial requirement of our technology. The requirement is natural because the provider of this very service must know what his/her service does, and must be able to answer queries about the current capabilities of his/her service. Usually, this functionality is implemented in some way on the side of the service provider, so that it may be also exposed as a part of the service. However, the problem is that this functionality must understand queries and answer them. Since these queries come from heterogeneous applications, and answers are sent also to a heterogeneous application, there must be a common language for expressing queries and answers. This very language is our description language DL.

The language is supposed to describes data, their types, attributes, locations, and processing by abstract functions, so that clients' requests can be expressed in this language. The language should also describe types of operation performed by services in terms of abstract functions they implement, as well as agents representing clients and services in terms of their goals, intentions, and commitments. These mental attitudes are essential for arranging services into workflow in order to realize the clients' requests. Hence, the language should also describes workflow formation and execution process, however, the language must be fully declarative, i.e., no explicit actions should be used.

Hence, there are three kinds of objects to be described in the language: Data (to be called resources in our framework), service-agents, and client-agents. For simplicity, from now on service-agent is called *service*, whereas client-agent is called *agent*. Since the language is supposed to be open and of distributed use, all names must be URIs (64). Therefore we must provide namespaces for resources, services, and agents.

3.2 Namespaces for Description Language

We assume that for any object, its name contains (as its inherent part) the communication address of this very object. Since communication address is dependent on transport protocol, an object can communicate using only one fixed transport protocol. This means, in turn, that all objects in our technology can use only one fixed transport protocol. This may be viewed as a drawback unless the protocol is simple and ubiquitous as it is in case of HTTP.

Resources are supposed to be passed from one service to another. Hence, the address of a resource is an URL pointing to the place on a www server where the resource is stored. The transport for resources is realized by the GET method of HTTP protocol.

Services and agents communicate by exchanging messages. The message format is defined at a high level of abstraction, i.e., at the level of logical communication between agents and services, so that the transport protocol is not essential and is not specified in the message format. Since the method POST of HTTP is chosen as the message transport protocol, the address format for agents' and services' addresses is the following:

```
http://host.name/path/party_id
```

Where `party_id` is the local identifier of the corresponding agent or service. However, in the case of agent, the string `party_id` is of the form `agentname/date-time` where `date-time` is a string of the type `xsd:dateTime`, and denotes the timeout set for the task delegated to the agent for realization.

Now, we are going to introduce a XLM-syntax of a simple version of first order predicate logic with types and without quantifiers. Although the syntax is typical for the first order logic, the evaluation of formulas will be done in the spatio-temporal manner. We will show that this language is sufficient to describe open and distributed environments consisting of heterogeneous components where clients' requests are realized and service components are used.

3.3 XLM-syntax of Description Language: upperEntish

Since the XML notation is long and cumbersome, a more convenient human readable notation of DL, that corresponds to the XLM syntax of DL, will also be introduced. Actually the notation resembles the standard notation of first order logic. For the complete XML syntax of DL we refer to the documents `formula.xsd` and `definitions.xsd` listed in the Appendix.

First of all we must introduce the XML format for names in our language DL. The following XML code defines the XML-type called `Concept`.

```
<xsd:complexType name="Concept">
  <xsd:sequence>
    <xsd:element name="shortName" type="xsd:string"/>
    <xsd:element name="longName" type="xsd:anyURI" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

It consists of two elements; the first one is called `shortName` and is of type `xsd:string`, and the second one is called `longName` and is of type `xsd:anyURI`. Hence, any concept has its own name that consists of a short name and a long name. The short name is just a string, whereas the long name is the pointer (URI) to the place where the concept was formally defined.

3.3.1 Type names

Now, let us go to the `definitions.xsd`, and see how the names for resource types, functions, and relations are introduced to DL. The general schema for defining type names in DL is the following.

```
 <xsd:element name="typeDefinition">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="definiens">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="typeName"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="definiendum">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="documentation"/>
```

```
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

As usually, formal definition consists of two basic elements: `definiens` and `definiendum`. The `definiens` contains the element `typeName` that is defined (in `formula.xsd`) as follows:

```
<xsd:element name="typeName" type="Concept"/>
```

That is, it is an instance of the XML-type `Concept`. The element `shortName` of the `typeName` contains a string, say `"Typ1"`. (Note that it is our convention that the first letter of the string is in upper case like in Java.) The element `longName` of the `typeName` contains a URL, e.g.,

```
   http://ii5.ap.siedlce.pl/Entish/my_type_def.xml#Typ1
```

This URL can not be arbitrary, i.e., `my_type_def.xml` is the very file name (being an instance of `definitions.xsd`) where this type name was defined, and the whole URL is the unique path (and pointer) to this very definition. This uniqueness is global, i.e., there cannot be any other place where this type was defined, because this very URL is an inherent part of the name of this type.

In our informal notation, the short name of this type is used, i.e., the name `Typ1`.

The second element of `typeDefinition` is `definiendum`. Since there are no type constructors in our language, this element contains only element `documentation` that may contain formal or informal description of this type.

Examples of type definitions can be found in `properEntish.xml` listed in the Appendix.

3.3.2 Relation names

Relation names are introduced to DL according to the following schema:

```
<xsd:element name="relationDefinition">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="definiens">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="relationName"/>
            <xsd:element ref="variable" minOccurs="0"
                                      maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="definiendum">
        <xsd:complexType>
```

```
          <xsd:sequence>
            <xsd:element ref="formula" minOccurs="0"/>
            <xsd:element ref="documentation"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The `relationDefinition` consists of two elements: `definiens` and `definiendum`. The first element, i.e., `definiens`, specifies the name (i.e., `relationName`) and the signature of the relation (i.e., the arguments of the relation) by means of the elements `variable` to be defined in Section 3.3.6. The element `relationName` is defined (in `formula.xsd`) as

```
<xsd:element name="relationName" type="Concept"/>
```

So that the element `shortName` of `relationName` contains the short name of the relation (say `rel1`), whereas the `longName` contains URL that is the unique pointer to the very XML file (being an instance of `definitions.xsd`) where this very relation was defined.

The `definiendum` of `relationDefinition` may consist only of the element `documentation` containing a more or less formal description of the relation. In this case the relation is primitive, and the precise way of evaluation of this relation MUST be described in the `documentation`.

If, however, the `definiendum` contains the element `formula` (defined in `formula.xsd` and explained in Section 3.3.7), then this element `formula` contains a formula that defines the relation. So that this relation is complex and is defined by primitive or less complex relations and functions that occur in the formula.

In our informal notation, the short name of this relation is used, i.e., the name `rel1` in the following way: `rel1( ?x, ?y )`, where `?x, ?y` are variables denoting the arguments of the relation.

Note that in any instance of `relationDefinition` described above as well as in any instance of `functionDefinition` described below, the variables (if there are any) occurring in the element `formula` (resp. element `term`) of the element `definiendum`, must be exactly the same as all variables occurring in the element `definiens`.

3.3.3 Function names

Function names are introduced to DL according to the following schema:

```
<xsd:element name="functionDefinition">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="definiens">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="functionName"/>
```

```
             <xsd:element ref="typeName"/>
             <xsd:element ref="variable" minOccurs="0"
                                     maxOccurs="unbounded"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="definiendum">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="term" minOccurs="0"/>
             <xsd:element ref="documentation"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
```

The `functionDefinition` consists of two elements: `definiens` and `definiendum`. The first element specifies the name (i.e., `functionName`) and the signature of the function (i.e., the arguments of the function, and type of the value returned by this function). The element `functionName` is defined (in `formula.xsd`) as

```
<xsd:element name="functionName" type="Concept"/>
```

So that the element `shortName` of the `functionName` contains the short name of the function (say `fun1`), whereas the `longName` contains URL that is the unique pointer to the very XML file (being an instance of `definitions.xsd`) where this very function was defined.

The `definiendum` of `functionDefinition` may consist only of the element `documentation` containing a more or less formal description of the function. In this case the function is primitive.

If, however, the `definiendum` contains the element `term` (defined in `formula.xsd` and explained in Section 3.3.4), then this `term` contains a term that defines this function. So that this function is complex and is defined by primitive or less complex functions and constants occurring in the term.

In our informal notation, the short name of this function is used, i.e., the name `fun1` in the following way: `fun1( ?x, ?y )`, where `?x, ?y` are variables denoting the arguments of the function.

Note that if a function has no arguments, then it is a functional constant, like the famous π number. It is important to grasp that functional constants are different from the element `constant` (defined in `formula.xsd` and described in Section 3.3.5) that contains the pointer to the specific data, and specifies the type of the data.

3.3.4 Terms

Terms are defined in the schema `formula.xsd` in the usual way, i.e.,

```
  <xsd:element name="term" type="Term"/>
```

```
<xsd:complexType name="Term">
  <xsd:sequence>
    <xsd:choice>
      <xsd:sequence>
        <xsd:choice>
          <xsd:element ref="variable"/>
          <xsd:element ref="constant"/>
        </xsd:choice>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element ref="functionName"/>
        <xsd:element ref="typeName"/>
        <xsd:element ref="term" minOccurs="0"
                                maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

So that every `constant` is a term, every `variable` is a term. Every expression, consisting of `functionName`, `typeName` denoting the type returned by this function, and several terms (as arguments of the function) of the types consistent with the function signature, is a term. So that the definition is recursive. The element `functionName` can not be arbitrary; it MUST already be defined in an instance of `definitions.xsd` available on www.

In our informal notation, `fun1( g( ?x ), cons1 )` is an example of a complex term.

3.3.5 Constants

A constant denotes specific data in our language. Constants are defined in the schema `formula.xsd` in the following way:

```
<xsd:element name="constant" type="Constant"/>
<xsd:complexType name="Constant">
  <xsd:sequence>
    <xsd:element name="constantName" type="Concept"/>
    <xsd:element ref="typeName"/>
  </xsd:sequence>
</xsd:complexType>
```

The element `constant` consists of `constantName` that specifies the name of the constant and `typeName` that specifies the type of the constant. The name consists of two elements: Obligatory `shortName` and optional `longName`. If there is no `longName`, then this very `shortName` contains the data that are denoted by this constant; in this case the data must be a literal or a string. Otherwise, the `longName` is a pointer (a URL) to a file where the data are stored.

In our informal notation, the contents of `shortName` is used to denote this constant.

It is important to notice that functional constants (i.e., function names with no arguments) and constants defined above are different notions.

3.3.6 Variables

Variables are defined in the schema `formula.xsd` in the following way:

```
<xsd:element name="variable" type="Variable"/>
<xsd:complexType name="Variable">
  <xsd:sequence>
    <xsd:element name="variableName" type="xsd:string"/>
    <xsd:element ref="typeName" minOccurs="0"
                                maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

A variable consists of `variableName` and optional `typeName` describing its type. If there is no `typeName`, then the type of the variable may be arbitrary. Otherwise the possible types are specified by one or several `typeName` elements.

3.3.7 Formulas

Formulas are defined in the schema `formula.xsd` in the usual way, i.e.,

```
<xsd:element name="formula" type="Formula"/>
<xsd:complexType name="Formula">
  <xsd:sequence>
    <xsd:choice>
      <xsd:sequence>
        <xsd:element ref="relationName"/>
        <xsd:element ref="term" minOccurs="0"
                                maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name="operator">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="or"/>
              <xsd:enumeration value="and"/>
              <xsd:enumeration value="implies"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element ref="formula" minOccurs="2"
                                   maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

So that a formula of our language is either an atomic formula or a complex formula. Complex formula is a disjunction, or conjunction or implication of formulas. Atomic formula consists of the element `relationName` (that MUST be already defined in an instance of `definitions.xsd` available on www), and several terms. The number of terms as well as their types MUST conform to the signature of the relation.

In our informal notation,

```
( psi1( ?x ) and psi2( ?y )) implies phi( ?x, ?y ) )
```

is an example of a formula.

It is important to note that a disjunction, conjunction, and implication operator may have more than two arguments. In the case of disjunction and conjunction it is natural because these operators are commutative. However, an implication consisting of more than two arguments, e.g.,

```
( phi1 implies phi2 implies phi3 ... )
```

is understood in the following way:

```
( phi1 implies ( phi2 implies ( phi3 ... )))
```

3.3.8 Evaluation of formulas

The crucial point of the logic we introduce is evaluation of formulas. It was already mentioned that introduction of new primitive relation name must be accompanied by the method of its evaluation specified in the `documentation` element of the relation definition. We introduce a special format for expressing a statement that a formula was true in a place and at a time. Since the formula evaluation is spatio-temporal, it means that the logic we introduce is also spatio-temporal.

The statement format (introduced in the schema `info.xsd`, see the Appendix) is the following:

```
<xsd:element name="signedInfo">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element ref="info"/>
       <xsd:element name="signature" type="xsd:anyURI"
                                         minOccurs="0"/>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

 <xsd:element name="info">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element ref="formula"/>
       <xsd:element name="place" type="xsd:anyURI"/>
       <xsd:element name="time" type="xsd:dateTime"/>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
```

The element signedInfo consists of the element info and an optional element signature that is a URL pointer to a file containing a signature of the element info. In the current version of the composition protocol (entish 1.0), the element signature is not used. The element info consists of three obligatory elements: formula, place, and time. The intended meaning of an info is that the formula was evaluated as true in this very place and at this very time. The element place contains URL pointer to the place, whereas time contains a string of type xsd:dateTime specifying the time of the evaluation.

This completes the introduction of upperEntish, that is, the syntax of DL. Now, we are going to introduce a collection of primitive types, relations and functions that may be seen as a proposal of upper ontology for service description.

Agents and services are represented by their states. Since the description language (we are going to introduce) is supposed to describe agents and services, it is reasonable to introduce the common format for agent and service state. This format is defined in the document state.xsd, see the Appendix.

3.4 State of agent and service

The element state is defined in the following way:

```
<xsd:element name="state">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="owner" type="xsd:anyURI"/>
      <xsd:element ref="goal"/>
      <xsd:element ref="intentions"/>
      <xsd:element ref="listOfCommitments"/>
      <xsd:element name="knowledge" type="listOfInfos"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

So that it consists of the following elements:

- owner that contains the address of the owner (i.e., an agent or a service) of the state.
- goal that contains the formula expressing agent's task (or the type of operation performed by a service if the state belongs to the service).
- intentions contains lists of agent's intentions.
- listOfCommitments contains a set of service's commitments.
- knowledge is the container for agent's/service's knowledge; it is a collection of elements signedInfo (i.e., evaluated formulas of the description language that were defined in the previous Section).

In the current version of the composition protocol if the state belongs to an agent, then the element listOfCommitments is not used, and if the state belongs to a service, then the element `intentions` is not used. However, it is also reasonable to consider agents that make commitments, as well as services that have intentions.

The element `goal` is defined as

```
<xsd:element name="goal">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="formIn" type="Formula" minOccurs="0"/>
      <xsd:element name="formOut" type="Formula"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

It consists of two elements: `formIn` and `formOut`. Each of them contains a formula of our description language. If the `state` belongs to a service, then the `goal` expresses the type of operation performed by the service, so that `formIn` contains a formula that describes the precondition of service invocation, whereas `formOut` contains a formula that describes the post condition (effect) of performing the operation by the service.

If the `state` belongs to an agent, then `formIn` is either empty or contains a formula that describes the precondition for realizing the agent's task, whereas `formOut` contains a formula that describes the agent's task.

The element `listOfCommitments` is defined as

```
  <xsd:element name="listOfCommitments">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="commitment" maxOccurs="unbounded"
                                              minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="commitment">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="formIn" type="Formula"/>
        <xsd:element name="formOut" type="Formula"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

It consists of elements each of which is called `commitment` and consists of two elements: `formIn` and `formOut`. Each of these elements contains a formula of our description language. `formIn` contains a formula that describes the precondition of the commitment, whereas `formOut` contains a formula that describes the post condition, i.e., the effect the service has committed to realize. Once a commitment is realized, it is removed from `listOfCommitments`, however the information about the realization is stored in `knowledge`. It is important to note that the format of service's `commitment` corresponds to the format of `goal` that is used to express the type of operation performed by the service.

The element `intentions` is defined as

```
<xsd:element name="intentions">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="plan" type="listOfFormulas"/>
      <xsd:element name="workflow" type="listOfFormulas"/>
      <xsd:element name="realized" type="listOfFormulas"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="listOfFormulas">
  <xsd:sequence>
    <xsd:element ref="formula" maxOccurs="unbounded"
                                        minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

So that it is composed of the following sub elements:

- `plan` is a sequence (list) of formulas (called intentions) describing agent's plan.
- `workflow` is a set of intentions (moved from `plan`) for which agent has already arranged commitments with services. They are supposed to form a workflow for realizing agent's task.
- `realized` is a set of intentions (moved from `workflow`) that have already been satisfied by realizing of the associated commitments.

An algorithm of agent functioning can be sketched as follows.

1. Agent's task is set as its first intention and put into `plan`.
2. Agent is looking for a service that can realize the first intention from its `plan`.
3. Once the agent finds out a service that has committed to realize its current intention, the intention is moved from `plan` to `workflow` whereas the pre-condition of the service's commitment is set as its new intention, and is put as the last element of the agent's `plan`.
4. If `plan` is an empty sequence, then the workflow for task realization is completed and may be executed.
5. Once the agent gets confirmation that an intention from `workflow` has been realized, it is moved to `realized`.
6. If `workflow` is empty, then the workflow has been already executed successfully, so that the task is realized, and the agent can send the final confirmation approving the transaction performed by the workflow.

The algorithm of service functioning is extremely simple. A service waits for agents. If an agent sends to the service a message with its intentions, and the intention can be realized by the service, then the service commits to realize this intention, so that appropriate formulas are put into the new element `commitment` of the service's state. Once the intention is realized, a confirmation is sent to the agent by the service, and the commitment is removed from the service's state.

3.5 properEntish as Description Language

The primitive types, relations, and functions that constitute together the description language are defined in the document `properEntish.xml` (see Appendix) as an instance of the schema `definitions.xsd`. Below, we describe them using our informal notation.

Primitive types:

- `Agent` is a primitive type; agent (i.e., element of type `Agent`) denotes a process equipped with its own state i.e., element `state` defined in `state.xsd` and explained in the previous Section. It is supposed that all essential data of an agent are stored in its `state`. Agent is dedicated for a single task realization. It is created when there is a task to be realized, and is terminated after the task realization or if the task cannot be realized.

- `Service` is a primitive type; service (i.e., element of type `Service`) denotes a process having its own state (i.e., element `state` defined in `state.xsd`). The main service's component is an application that processes data. Processing data may result in influencing the real world, e.g., purchasing a commodity or withdrawing some amount of money from a bank account, or just performing some physical actions like switching off/on a washing machine.

- `Time`; element of this type is a string written according to `xsd:dateTime` format.

- `Token`; element of this type is an arbitrary string. It is used as value of function `token( ?resource )`. Tokens serve to identify resources at the language level. Note that in our language, resources are only described so that their format may be arbitrary, e.g., XML, MS Word, txt, binary, and so on.

- `Index`; element of this type is a string of the form of decimal numbers separated by dots in the very same way as IP addresses, e.g., `0.12.3` or `34.0.11.45.1` or `5.34` It is used as value of function `index( ?resource )`. The value (an index) is associated with an option during workflow formation. Indexes serve to determine interdependencies between options that describe resources in a workflow.

Primitive relations:

- `timeout( ?t )` can be evaluated at any host. It is true if the time `?t` is later (bigger) than the current GMT time at the host.

- `( ?x=?y )` is a polymorphic equality relation. It can be evaluated if `?x` and `?y` are of the same type.

- `isIn( ?resource, ?service )` states that `?resource` is in `?service`. It can be evaluated locally only by `?service`.

- `intentions( ?agent )` is an atomic formula. It is evaluated only locally by `?agent.` During an evaluation it is replaced with the disjunction of all formulas from the element `plan` of the `state` of the `?agent`.

- `formInOperationType( ?service )` is an atomic formula to be evaluated only by `?service`. During an evaluation it is replaced with the formula from the element `formIn` of `goal` of the `state` of `?service`. The formula describes the precondition necessary for `?service` invocation.

- `formOutOperationType( ?service )` is an atomic formula to be evaluated only by `?service`. During an evaluation it is replaced with the formula from the element `formOut` of `goal` of the `state` of `?service`. The formula describes the post condition of `?service` invocation, i.e., the result of performing the operation by `?service`.

- `formInCommitment( ?service )` is an atomic formula evaluated only by `?service`. During an evaluation it is replaced with the disjunction of formulas from the `formIn` elements of all `commitment` elements of `listofCommitments` of the `state` of `?service`. It describes the preconditions of the commitments made by `?service`.

- `formOutCommitment( ?service )` is an atomic formula evaluated only by `?service`. During an evaluation it is replaced with the conjunction of formulas from the `formOut` elements of all `commitment` elements of the `listOfCommitments` of the `state` of `?service`. It describes the post conditions of the commitments made by the `?service`.

- `false` is the atomic formula that is always false.

- `true` is the atomic formula that is always true.

It is worth to note, that the primitive relations: `intentions`, `formInOperationType`, `formOutOperationType`, `formInCommitment` and `formOutCommitment` of the description language correspond to some elements of `state`.

There are only two predefined functions that serve to identify resources described during workflow formation.

Primitive functions:

- `token( ?resource )` is the function that returns a token (an element of type `Token`) determined for `?resource` by the service that expects it to be delivered as its input.

- `index( ?resource )` is the function that returns an index (element of type `Index`) that is associated with an option. Options describe the resources in a workflow. Indexes serve to determine interdependencies between the options in the workflow.

A formula of our description language is syntactically valid (well formed) if it is constructed according to the syntax specified in `formula.xsd` and `properEntish.xml`, and if all the names of types, relations, and functions occurring in the formula have already been defined in XML documents that are instances of `definitions.xsd`, and the documents are available by HTTP.

The language specified above is called Entish.

3.6 Entish and the meaning

Entish is supposed to have precise and machine processable semantics. In order to explain this semantics, let us fix our attention on the concept of meaning (i.e., semantics) for a while.

3.6.1 Meaning

This section is based on the discussion concerning the meaning, that took place at ws-www@w3.org mailing list (subject: Meaning, URL http://lists.w3.org/Archives/Public/www-ws/) in May and June 2003.

Language alone is merely a syntax, so that formal inference and axioms (i.e., a naming convention and some rules how to transform one string onto another) have nothing (or rather little) to do with the meaning of a language.

According to the Wittgenstein's thesis (25), the meaning of language is in its use. Hence, the meaning is realized through a consensus on how to use the primitive concepts of the language. The consensus is a never ending process; some new useful concepts are being introduced (according to some fixed conventions) whereas some useless or old ones disappear. New complex notions are created as abstractions of the existing ones. The crucial point is to start the process from the very roots, i.e., from the really primitive concepts and to specify explicitly the abstraction rules.

Let us present here some prominent examples of abstract notions.

- Turing machine representing computable functions; it is an abstraction from the number of steps needed to halt and from the length of the tape.
- Continuity; it is an abstraction from the resolution, i.e., number of pixels per square unit.
- infinite sets, e.g., the set of natural numbers is an abstraction from counting process.
- most of the relations are abstractions from situations resulting from performing some operations.

Usually, the meaning of such abstract notions can be characterized by axioms in terms of other notions. However, it seems that this very meaning of an abstract notion can be reconstructed by decomposing it into more and more primitive ones so that finally it may be reduced to really primitive concepts connected by abstraction rules.

Since it is not the right place to discuss the fundamental problems of semantics, let us only mention that important theoretical contribution to the problem of semantics was done by the great philosophers and logicians to mention only Immanuel Kant (13), Edmund Husserl (12), Charles Peirce (18), Ludwig Wittgenstein (25), Felix Kaufmann (14), Alferd Tarski (23), and Jerzy Łoś.

3.6.2 Entish as an open language

Let's go back to Entish. Openness of Entish is clear because of the format of names. Anyone can introduce his/her own concepts to the language, however it must be done according to the fixed rules, that is, the concept must be defined in a document being an instance of the schema `definitions.xsd`, and the document must be available on the www. So that the name of a new concept contains the pointer to the document where this very concept was defined.

3.6.3 Meaning of Entish

Generally, Entish describes applications that process and exchange data. An application is described either as an agent in terms of its intentions, or as a service in terms of the precondition and the postcondition of the operation it performs as well as its commitments. Data are described as resources of some types. Data processing is described by functions, whereas data passing is described by relations. The crucial point is that in order to describe data processing and exchange in Entish, the concepts used in the description must be defined according to the rules determined by the schema `definitions.xsd`.

Since axioms were not even mentioned, the question is what the meaning of the language is. How do the names get their meaning in this language? The answer is extremely simple. A name gets its meaning by pointing to the place where it was originally defined either as a primitive concept or as a complex concept. This very name contains the pointer to this original place. There are two cases:

- If the name denotes a primitive concept, then the definition consists only of definiens. Since the name of this primitive concept is unique, and points to the very unique place where it was originally defined (only by definiens) the meaning of this primitive concept is fixed. Anyone who uses this name refers to this very unique place where it was originally defined. Informal meaning of this concept may be described in a natural language also in this place.

- If the name denotes a complex concept, then its meaning is defined in the definiendum by means of names of less complex concepts. Any of these less complex concepts is either a primitive concept (with meaning fixed by its name) or is defined in terms of less less complex concepts. The result of this unfolding (i.e., substituting name of complex concept by its definiendum) is a tree whose root is the name, the nodes are names of less complex concepts, and the leaves are names of primitive concepts.

There are three kinds of primitive concepts that can be introduced to Entish, namely, types of resources, functions, and relations. It is clear that there should be no constrains for introducing names for new primitive resource types and primitive functions, because this is related to introducing new services that perform new operations on new data. However, in the case of primitive relations it is natural to constrain primitive relations to a fixed small set. The reason is as follows.

What the language describes is merely data processing and exchange that can be described in terms of resource types and functions. Hence, except for a small number of primitive relations (needed to express equality, the fact that a resource is in some place, etc.,) there is no need to introduce more primitive relations. This is important for keeping the semantics of Entish clear and precise. For this very reason it is required that any primitive relation introduced to Entish must be accompanied by precise specification how to evaluate this relation in an automatic way. Without this requirement, the only way to provide meaning for this primitive relation is to characterize it by axioms. This, however, would be a return to the classic formal semantics.

Hence, Entish can be developed by defining and using, step by step, more and more sophisticated concepts having their meaning maintained precise by keeping chains of unique names that end up with unique names of primitive data types and functions.

This may be seen as a trick, nothing new, and no (machine readable) semantics at all. However, it works. The critical point is to grasp who can create formulas and who (rather what) processes them. In fact, the semantics, or language understanding is always on the side of humans, i.e., users and service providers. They express their requests, and types of operations performed by their services in terms of data types and functions. Only they can create formulas, so that they must understand what the formulas mean. In the composition protocol, we are going to introduce, the formulas are processed automatically (by applications) in the very similar way as data are.

3.6.4 Entish - say nothing that isn't worth saying

Now, let us explain why our description language is called Entish.

Original Old Entish was the ancient language of Ents, see (24) *The Lord of the Rings* by J.R.R. Tolkien. Tolkien describes Entish as *agglomerated* and *long-winded*. This was due to the fact that each *word* was actually a very long and very detailed description of the thing in question. Treebeard said of his own Entish name that it was *growing all the time, and I've lived a very long, long time; so my name is like a story. Real names tell you the story of the things they belong to in my language, in the Old Entish as you might say.*

Treebeard's own description of Entish is as follows:

It is a lovely language, but it takes a very long time to say anything in it, because we do not say anything in it, unless it is worth taking a long time to say, and to listen to.

The analogy with the language of Ents is obvious. The names in our description language DL can be compared to the real names of the Old Entish, that is, a name of DL has the precise and clear meaning that is associated with this very name. This meaning can be unfolded by referring to the names of really primitive concepts. However, the process of unfolding may sometimes be complex and take a time.

3.7 Entish and the Semantic Web - a comparison

Let us introduce the idea of Semantic Web by the following citations from (55) and (10):

"The Semantic Web is the representation of data on the World Wide Web. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF) (56) which integrates a variety of applications using XML for syntax and URIs for naming."

"The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

There is also a number of more or less formal introductions of the idea of the Semantic Web, see for example (57), (58), and (59). The short overview of the Semantic Web presented below is based on the contents of the website (58).

The Semantic Web is an evolving collection of knowledge, built to allow anyone on the Internet to add what they know and find answers to their questions. Information on the Semantic web, rather than being in natural language text, is maintained in a structured form (RDF) which is fairly easy for both computers and people to work with.

The problem is that, in Semantic Web, concepts from natural language are used as well as their intuitive semantics. In Entish, the meaning of concepts is built from the very beginning starting with the primitive concepts, and according to the precise rules.

3.7.1 Description in the Semantic Web

An RDF statement is very much like a simple sentence, except that almost all the words are URIs. Each RDF statement has three parts: a subject, a predicate and an object. Let's look at a simple RDF statement:

```
<http://www.ipipan.waw.pl/mas/stan/>
<http://love.example.org/terms/reallyLikes>
<http://www.w3.org/People/Berners-Lee/Weaving/> .
```

The first URI is the subject. In this instance, the subject is the person Stanislaw Ambroszkiewicz. The second URI is the predicate. It relates the subject to the object. In this instance, the predicate is "reallyLikes." The third URI is the object. Here, the object is Tim Berners-Lee's book "Weaving the Web." So the RDF statement above says that Stanislaw Ambroszkiewicz really likes "Weaving the Web." Although the statement is true, it does not mean that he agrees with the thesis stated in the book.

It is rather curious to identify a person with one of his / her personal websites. The same may be applied to the subject that is supposed to denote a book written by T. Berners-Lee. As to the predicate, its meaning is supposed to be fixed by pointing to existing or not existing HTML document.

RDF statements can say practically anything, and it doesn't matter who says them. There is no official website that says everything about Weaving the Web. This leads us to an important RDF principle, namely "anything can say anything about anything". Information is spread across the Web, and two people can even say contradictory things – Bob can say that Aaron loves Weaving the Web and John can say that Aaron hates it. This is the freedom that the Web provides.

The statement above is written in N-Triples, a language that allows you to write simple RDF statements. However, the official RDF specification defines an XML representation of RDF, which is a bit more complicated, but says the same thing:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:love="http://love.example.org/terms/"
>
    <rdf:Description rdf:about="http://www.ipipan.waw.pl/mas/stan/">
         <love:reallyLikes rdf:
         resource="http://www.w3.org/People/Berners-Lee/Weaving/"/>
    </rdf:Description>
</rdf:RDF>
```

Hence, in Semantic Web the basic component for description is the so called ”triple” consisting of Object URI, Predicate URI, and Subject URI. And using these triples anything can say anything about anything. It seems that this means exactly the same as ”nothing can say nothing about nothing”.

Entish is a simple version of a language of first order logic with types, so that its formulas are used for description. The way the new names (denoting concepts) are introduced to Entish is precisely defined. Entish is a language describing only how applications process and exchange data. The names for the data types and constants denoting concrete data, and the names for operations types performed by the applications, as well as the names for relations used in the description, had to be introduced to the language. Although, data processing and exchange may cause some effects on the real world, concepts from natural language cannot be used in Entish directly. A name from natural language can be, of course, introduced to Entish only as a string in the element short name of the concept name. However, the long name of this very concept name must be the pointer to the XML document where this concept was defined along with its meaning.

3.7.2 Names in the Semantic Web

On the Semantic Web, each thing (and relationship) is identified using more complicated unambiguous names. The names are web addresses (sometimes called URLs or URIs). This can cause some confusion, e.g., Is ”http://www.ibm.com” a company or the company’s web site?

There are also special names for text strings (literals), and temporary names are allowed which function like pronouns. This lets us write ”The country with the ISO country code ’US’ has a president who has the name ’George W. Bush’” without ever using an identifier for the US.

In Entish the names are in fact also URI, that is, an Entish name (except constants that are literals) contains a pointer (URL) to the XML document where this very name was defined.

3.7.3 Logic in the Semantic Web

Some facts (like "Mary is a mother" and "A mother is a kind of parent"") lead logically to other facts ("Mary is a parent"). That is often intuitive to people, but can be very hard to explain to a computer. When properly programmed, however, computers can be very helpful in figuring out which facts follow logically from other facts.

A precise explanation of one's terms and reasoning in some subject area, which can allow computers to help, is called an ontology. Ontologies can be expressed in various languages and carried by the Semantic Web. With them, computers can sometimes act as if they "understand" the information they are carrying. This is where the term "semantic" comes in.

No logic languages have yet been recommended for the Semantic Web. Some of the experimental languages are RDF Schema (60), DAML (61), and OWL (62).

Entish is a well defined logic language, so that there is no problem with automatic reasoning. There are no ontologies in the classic sense, i.e., as a formal specification of concepts by axioms. In Entish, an ontology may be understood as a collection of concepts that correspond to one application domain. So that such ontology may include names of types, their attributes, names for functions operating on these types as well as names for relations. It is important to stress once again that a name for relation can be introduced either as a complex concept, or as a primitive concept, however, under the condition that a precise evaluation of the primitive relation must be provided in its definition. One more important feature of Entish is that its formulas are evaluated locally and temporarily, i.e., in spatio-temporal manner, in a certain place and at a certain time.

3.7.4 Services in the Semantic Web

The process of buying a book over the Semantic Web may be described in the following way:

1. You browse/query until you find a suitable offer to sell the book you want.

2. You add information to the Semantic Web saying that you accept the offer and giving details (your name, shipping address, credit card information, etc). Of course you add it with access control so only you and seller can see it, and you store it in a place where the seller can easily get it, perhaps the seller's own server, and you notify the seller about it.

3. You wait or query for confirmation that the seller has received your acceptance, and perhaps (later) for shipping information, etc.

This approach allows automation of the process, detailed record-keeping, and excellent process abstraction. However, this automation has not been realized so far.

It seems that this very automation is extremely hard to realize. We propose quite simple and already implemented protocol for realizing clients' requests. It is based on Entish, so that a request is realized by discovering, and composing services. Our protocol realizes the functionality of the scenario presented above as well as much more.

Now, we are going to present some details of the composition protocol starting with the format of message.

3.8 Message format

The service composition protocol, we are going to propose, is a conversation protocol between agents, services, and infoServices. During a conversation (i.e., a protocol session) messages, that may affect the `state` of sender and recipient, are exchanged. The language of the message contents is supposed to be our description language, i.e., the contents is an evaluated formula of our language describing a situation between an agent, services and resources. The `message` format defined in the document `message.xsd` (see the Appendix) is extremely simple.

```
<xsd:element name="message">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="header">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="from" type="xsd:anyURI"/>
              <xsd:element name="to" type="xsd:anyURI"/>
              <xsd:element name="protocolName"
                                         type="xsd:string"/>
              <xsd:element ref="protocolVersion"/>
              <xsd:element name="protocolSession"
                                         type="xsd:string"/>
              <xsd:element ref="protocolOrder"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="body" type="listOfInfos"/>
     </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

So that it consists of the following elements:

- `header`:
  - `from`; it contains sender's address.
  - `to`; it contains recipient's address.
  - `protocolName`; it contains the name of protocol.
  - `protocolVersion`; it contains the version of the protocol.
  - `protocolSession`; it contains the session identifier.
  - `protocolOrder`; it contains the type (order) of the message in the protocol.
- `body`; it is the contents of the message.

Although we do not use SOAP, the `message` format was designed to fit to the format of SOAP envelope. That is, our `header` corresponds to the header of SOAP, and our `body` corresponds to the body of SOAP.

The element `body` contains facts, i.e., evaluated formulas. The format of evaluated formula is `info` defined in the document `info.xsd`, and is composed of the following elements: `formula, time, place`, and `signature`. The intended meaning of `info` is that `formula` was true at the `time`, in the `place`, and this was stated by the one who made the `signature`.

3.9 Composition protocol entish 1.0

Now, we are ready to explain the idea of our protocol for service composition; formal specification is presented in Chapter 6.

Generally, a protocol is a specification of message exchange between parties, and how sending or receiving a specific message type changes the `state` of sender and recipient. The parties that participate in a conversation (protocol session) are agent, services, and infoService. The format of agent and service `state` was specified in Section 3.4. The state format of infoService is extremely simple; it consists of a collection of facts, i.e., of elements of type `info`.

The protocol is divided into the following conversation phases: Publication, discovery, workflow formation, workflow execution, and finally the phase realizing a distributed transaction. The publication phase is a conversation between service and infoService. The discovery phase is between agent and infoService. The workflow formation phase is between agent and services. The workflow execution and the transaction phase are between agent and the services engaged in the workflow.

The protocol serves for the following purposes:

1. Applications could be joined to our infrastructure as services.

2. Tasks, issued by the clients, could be realized.

Although tasks and intentions could be defined as arbitrary formulas of the description language, we define the canonical formats of the following kinds of formulas: task formula, intention formula, commitment formula, and formula describing the type of operation a service performs. These formats are specified in Chapter 6.

Let us present a sketch of the workflow formation phase that consists of service invocation and service composition. Service invocation comprises the following five steps:

1. An agent sends to the service0 the message: *"my intention is psi0 "* , formally it is the following formula

   ```
       ( psi0 implies intentions( agent ) )
   ```

2. The service0 responds with the following commitment: *"I commit to realize psi0 if psi1 is satisfied"*, formally:

   ```
   (
   (psi1 implies  formInCommitment(service0))
   ```

```
 and
(formOutCommitment(service0) implies psi0)
)
```

It is supposed that the commitment can be realized, i.e., it is consistent with the operation type performed by the service0. It means that once the formula `psi1` has been satisfied, then the precondition for the service0 invocation is satisfied, and the operation can be performed by the service0. Once the operation has been performed (i.e., the post condition of the operation type has been satisfied), then the formula `psi0` is satisfied.

3. Suppose that the formula `psi1` is satisfied by another service1.

4. Then, the formula `psi0` is satisfied, and the agent's intention is realized.

5. Finally, the service0 sends a confirmation to the agent.

Composition of two services (service0 and service1) is arranged by the agent in the following way. The agent arranges the realization of its first intention `psi0` by the service0. The service agrees to realize this intention conditionally, i.e., if the formula `psi1` is satisfied. Then, the agent puts the formula `psi1` as its current intention, and looks for another service that could realize this intention. Suppose that the agent got to know that it follows from the operation type of the service1 that the service could realize its current intention. The agent starts a conversation with the service1 by sending the message: *"my intention is* `psi1` *"* . Once the service1 agrees to realize this intention, the operations of the service0 and the service1 are composed, and form a part of a workflow the agent must construct in order to realize its task.

In the protocol entish 1.0, the message elements are specified as follows.

- `protocolName` (i.e., the protocol name) is set as `entish`.
- `protocolVersion` is set as `1.0`.
- `protocolSession` is set as the address (see Section 3.2) of the process (agent or service) that has initialized the conversation session. There are two cases. A newly created agent can initialize a session dedicated to its task realization; the session last as long as the agent exists. Or a newly created service initializes a session for publishing the type of operation it performs; the session lasts as long as the service exists.
- `protocolOrder` denotes the message type (order) in the protocol.

We distinguish the following message types:

- `protocolOrder` is set as 000. Then, service sends to infoServices an `info` containing formula expressing its operation type. The contents, i.e., the `info`, may be forwarded in a messages having the same order and session.
- `protocolOrder` set as 111. Then, infoService sends message of this order, as the reply to the service, confirming the publication of the service's operation type, and setting a timeout for its validity. The `info` contains the timeout

formula. The infoService can also send message of this order to an agent as the reply to the agent message of order 001. The `info` of this message is an `info` sent previously by a service. These two messages have different `protocolSession` elements; the first one contains the address of service, whereas the second one contains the address of the agent.

- `protocolOrder` is set as 001. Then, agent sends `info` with formula expressing its current intention to an infoService or to a service. The contents, i.e., the `info`, may be forwarded in a messages having the same order and session.

- `protocolOrder` is set as 021. Then, a service sends to agent an `info` with a formula expressing its commitment to realize the agent's intention sent in a previous message of `protocolOrder` set as 001.

- `protocolOrder` is set as 020. Then, agent sends `info` to the service and cancels the commitment from the previous message. The message contents, i.e., the `info`, is the same as the contents of the previous message of `protocolOrder` set as 021. This message can be sent only if the message of `protocolOrder` set as 222 has not been sent by the agent before.

- `protocolOrder` is set as 002. Then, service sends `info` to the agent to inform that the commitment (the service has made towards the agent) is canceled. The message contents is the same as the contents of the previous message of `protocolOrder` set as 021. This message can be sent only if the message of `protocolOrder` set as 222 has not been sent by the agent before.

- `protocolOrder` is set as 222. Then, agent sends (synchronously!) confirmation to all the services engaged in the workflow, informing them that the workflow is completed and ready to start, i.e., there are no elements in the agent's `plan`. The contents of the message is `info` with the formula `( true )` . After sending it successfully, the workflow execution is started. For the services that provide initial resources in the workflow, it is the signal that they may send messages of `protocolOrder` set as 321 to the next services in the workflow.

- `protocolOrder` is set as 321. Then, service0 sends `info` about the resource (it has already produced) to the next service1 in the workflow. The next service1 is supposed to download the resource. The `info` contains formula of the form

  ```
  (
    ?res1=url1
    and
    token(?res1)=tok1
  )
  ```

- `protocolOrder` is set as 331. Then, the service1 in the workflow (after successful downloading the resource) sends `info` to the service0 confirming the downloading. The `info` contains formula of the form

```
(
  isIn(?res1, service1 )
  and
  token(?res1)=tok1
)
```

- `protocolOrder` is set as 333. Then, the service0 sends the confirmation to the agent. The contents of the `info` is the post condition of the commitment made by the service to the agent. It is the intention formula sent in the message of `protocolOrder` set as 001.
- `protocolOrder` is set as 009. Then, the message is sent by a service (participating in the workflow) to the agent if the service cannot perform the operation it has declared to do in the commitment that it made in message of `protocolOrder` set as 021. This message can be sent only in the workflow execution phase, i.e., after sending the message of `protocolOrder` set as 222 by the agent.
- `protocolOrder` is set as 090. Then, agent sends to all services engaged in the workflow that the transaction is canceled. The contents of the message is `info` with the formula `( false )`. The services perform rollback, i.e., undo the effect resulting from operation performance.
- `protocolOrder` is set as 999. Then, agent sends (synchronously) confirmation, to all participants of the workflow, informing that the workflow has been successfully executed and the result is approved. The message may be sent if `plan` and `workflow` in agent's `state` are empty. The contents of the message is `info` with the formula `( true )`. This message type implements successful realization of the distributed transaction.

It is important to note that the message orders defined above correspond to the performatives of KQML (26) and FIPA ACL (27).

In the next two Chapters, detailed description of the implementation of the running examples (see Chapter 2, Section 2.3) are presented. They serve as an explanation of the composition protocol. A detailed specification of the composition protocol is presented in Chapter 6.

3.10 Chapter summary: What is enTish?

enTish is a proposal of an experimental technology for describing and composing heterogeneous services in open and distributed environment. enTish is merely a formal specification of description language Entish, and non-formal but precise specification of composition protocol entish 1.0.

Abstract architecture for implementing enTish is proposed in Chapter 7. Now, enTish is in the stable version 1.0 verified by several implementations.

Chapter 4

Implementation of the First Running Example

Let us reconsider the first running example from Chapter 2. The example is about flight booking and the travel agent FirstClass. This chapter is devoted to the description of an implementation of this example explaining some details of the service composition protocol entish 1.0 that was specified formally in Chapter 6.

4.1 Service operation and state

In order to implement the example we must specify the resource types (for orders and invoices) and the function that given an order produces an invoice.

First of all we must look at the contents of EntishDictionary that is the central repository (implemented in the enTish prototype) of instances of the schema `definitions.xsd`. We must find out if the names for types of resources, names of their attributes and the name for the function that the FirstClass has implemented as its operation have been already defined there. If they have not been defined there, then we must define them using either EntishDictionary or manually creating a XML-document being an instance of `definitions.xsd`.

For the realization of the service First Class, we need the following types:

- `Order`, that defines air ticket orders. Let `?order` be a variable of type `Order`. This type has the following attributes:
  - `person( ?order )`, returns string that specifies person name.
  - `creditCard( ?order )`, returns string that specifies credit card details.
  - `airLine( ?order )`, returns string that specifies airline.
  - `destination( ?order )`, returns string that specifies flight destination.
  - `departure( ?order )`, returns string that specifies flight departure.
  - `price( ?order )`, returns string that specifies price.
- `Invoice`, that defines air ticket invoices. Let `?invoice` be a variable of type `Invoice`. This type has the following attributes:
  - `Destination( ?invoice )`, returns string that specifies flight destination.
  - `Departure( ?invoice )`, returns string that specifies flight departure.
  - `Price( ?invoice )`, returns string that specifies ticket price.
  - ... and perhaps some others attributes.

We define the following function: `SELL-TICKET: Order -> Invoice` which, given an order, produces an invoice. It describes production of air ticket invoice from air ticket order.

Suppose that there is a service called FirstClass which is a travel agent office. The service implemented the function `SELL-TICKET` as its operation. The service FirstClass has its `state`. The formula

```
isIn( ?order, FirstClass )
```

is put into the element `formIn` of `goal` of the `state` of the service. The formula

```
    isIn( ?invoice, ?anyService )
    and
    ?invoice = SELL-TICKET( ?order )
```

is put into the element `formOut` of `goal` of the `state` of the service. Once these two formulas are put into `formIn` and `formOut` of `goal` of the `state` of FirstClass, they describe the type of operation performed by the service FirstClass. That is, if a resource of type `Order` is delivered to the FirstClass, then the FirstClass produces an appropriate resource of type `Invoice` that can be delivered to any service (place) by the FirstClass.

4.2 Publication of the operation type of FirstClass

If a service wants to publish its operation type to an infoService, then it sends `message` of `protocolOrder` 000, and `protocolSession` set as FirstClass' address. The `info` is created by the FirstClass, so that the address of FirstClass is put into the element `place` of the `info`. The element `formula` of `info` contains the following formula:

```
( ( isIn( ?order, FirstClass ) implies
    formInOperationType( FirstClass )
  )
  and
  ( formOutOperationType( FirstClass ) implies
    ( isIn( ?invoice, ?anyService ) and
      ?invoice = SELL-TICKET(?order)
    )
  )
)
```

Note that this formula expresses the operation type of the service FirstClass. If the infoService agrees to publish the operation type of FirstClass, then the infoService puts the `info` into the element `knowledge` of its `state`.

Then, infoService replies with the `message` of `protocolOrder` 111, with the same `protocolSession`, and with `info` (created by infoService) with the formula:

```
    timeout(t1)
```

that sets the timeout for the validity of the entry (i.e., the `info` sent by FirstClass) in the infoService registry.

4.3 Task

Client's task is the following:
Purchase airline ticket for a flight from Warsaw to Geneva; departure: Warsaw, June 22, 2002; timeout for task realizing: July 1, 2002.

The client is associated with a user interface called TaskManager. The client's task is formulated as follows:

```
(3.1)
      (  timeout("July 1, 2002")
         and
         isIn( ?invoice, TM-00)
         and
         ?invoice = SELL-TICKET( ?order )
         and
         token( ?invoice ) = tok1
         and
         (
          index( ?invoice ) = "0"
          and
          Destination( ?invoice )="Geneva"
          and
          Departure( ?invoice )="Warsaw, July 22, 2002" )
         )
       )
```

The meaning of the formula is that the resource `?invoice` is to be delivered to the service TM-00 (associated with the TaskManager) by July 1, 2002; the `?invoice` is the output of some operation that implements the abstract function `SELL-TICKET` describing production of airline ticket invoice from airline ticket order. The resource `?invoice` is specified by some attributes like `Destination` and `Departure`. The functions `index` and `token` are polymorphic functions defined for all types of resources. They are introduced in `properEntish.xml`, see the Appendix. The role played by these functions in the composition protocol will be explained later on.

The TaskManager must create a service TM-00 for storing the final resource specified in the task formula. In order to do so the following formula:

```
(3.2)
      (  timeout("July 1, 2002")
         and
         isIn( ?invoice, TM-00)
         and
         token( ?invoice )=tok1
         and
         (
          index( ?invoice ) = "0"
          and
          Destination( ?invoice )="Geneva"
          and
          Departure( ?invoice )="Warsaw, July 22, 2002" )
         )
      )
```

is put into the element `formIn` of a `commitment` of the `state` of the service TM-00. It means that the service TM-00 is expecting that a resource specified in this formula will be delivered to it.

The TaskManager creates also a service for delivering initial resource needed for the task realization. The service name is TM-01. The formula `( true )` is put into the element `formIn` of a `goal` of the `state` of the service TM-01. Whereas the formula `isIn( ?order, ?anyService )` is put to into the element `formOut` of the `goal` of the `state` of the service TM-01.

These two auxiliary services are created dynamically by the TaskManager. The service TM-00 is supposed to receive and store the final resource of type `Invoice` specified in the task, i.e., the resource with the token `tok1`. Whereas the service TM-01 is supposed to deliver initial resource of type `Order`. This initial resource is supposed to be created by the client associated with the TaskManager.

4.4 Agent

The task formula (3.1) is delegated to the agentServer (see the Chapter 7 for the details of abstract implementation architecture) where a process is created that is responsible for a realization of this task. The process is called agent01 and has its own `state`. The task formula (3.1) is put into the element `formOut` of `goal` of the `state` of the agent01.

The new element `info` is created by TM-01. The `info` contains the formula that states that TM-01 produces resources of type `Order`. It is the following formula:

```
(4.1)
      ( true implies formInOperationType( TM-01 ) )
      and
      ( formOutOperationType( TM-01 ) implies
        isIn( ?order, ?anyService)
      )
```

The `info` is put into the element `knowledge` of the agent01's `state`.

Agent01 starts its algorithm. Initially, the element `plan` of `intentions` of agent01's `state` is empty. Because the function composition in the task formula (3.1) is simple, i.e., only one function occurs there, this task formula becomes the first agent01's intention. Let `int0` denotes the following formula:

```
(4.2)
      (  timeout("July 1, 2002")
         and
         isIn( ?invoice, TM-00)
         and
         ?invoice = SELL-TICKET( ?order )
         and
         token( ?invoice ) = tok1
         and
         (
          index( ?invoice ) = "0"
          and
          Destination( ?invoice )="Geneva"
          and
          Departure( ?invoice )="Warsaw, July 22, 2002" )
         )
       )
```

The formula `int0` is put as the first element of `plan` of `intentions` of the agent01's `state`.

For the general way to create the first agent's intention from a task formula, see Chapter 6.

4.5 Workflow construction

The `protocolSession` of all the messages below is set as agent01's address.

Message of `protocolOrder` 001, with the `info` (created by agent01, i.e., the agent01's address is put into the elment `place` of the `info`) that contains the formula:

```
    int0 implies intentions( agent01 )
```

is sent by the agent01 to an infoService (suppose that there is at least one). The intended meaning of the formula is that satisfaction of the formula `int0` is an intention of agent01.

Suppose that infoService replies by forwarding the `info` of the message with `protocolOrder` 000 sent previously by the service FirstClass. The message sent by infoService to the agent01 has the `protocolOrder` 111, and the `protocolSession` set as the agent01's address. Once the message is delivered to the agent, agent01 puts the `info` into its `knowledge`. So that agent01 knows that (according to the infoService) the service FirstClass can realize its current intention. Hence, agent01 sends the message with `protocolOrder` 001 to FirstClass. The `info` of this message is the same as the one sent by agent01 to the infoService in the previous message.

Suppose that the service FirstClass commits to realize the `int0`, however, under the condition described by the following formula:

```
(5.2)
   (
     timeout("June 30, 2002")
     and
     isIn(?order, FirstClass)
     and
     token(?order)=tok2
     and
     (
        ( index( ?order )="0.0" and
          price(?order)="300euro" and airLine(?order)="Swissair"
          and
          destination(?order)="Geneva"
          and
          departure(?order)="Warsaw, July 22, 2002"
        )
        or
        ( index( ?order )="0.1" and
          price(?order)="350euro" and airLine(?order)="Lufthansa"
          and
          destination(?order)="Geneva"
          and
          departure(?order)="Warsaw, July 22, 2002"
        )
        or
        ( index( ?order )="0.2"
          and
          price(?order)="280euro" and airLine(?order)="LOT"
          and
          destination(?order)="Geneva"
          and
          departure(?order)="Warsaw, July 22, 2002"
        )
     )
  )
```

Let the formula (5.2) be denoted by `pre`. The meaning of `pre` is that `?order` with token `tok2` is delivered to FirstClass by June 30, 2002, and attributes of the `?order` should be chosen according to the ones listed in this formula. Actually, this attribute listing represents the options identified by indexes. The first option is identified by index "0.0", the second one by index "0.1", whereas the third one by the index "0.2". Note that the `token` as well as `index` of the input resource `?order` are determined by the service FirstClass that made this commitment.

The `state` of FirstClass is changed. A new element `commitment` of `listOfCommitments` of its `state` is created, and the formula `int0` is put into the element `formOut` of the `commitment`, whereas the formula `pre` is put into the element `formIn` of the `commitment`.

The service FirstClass replies to agent01 with a message with `protocolOrder` 021 with `protocolSession` set as agent01's address and with `info` (created by FirstClass) having the formula:

```
( pre implies formInCommitment( FirstClass ) )
and
( formOutCommitment( FirstClass ) implies int01 )
```

The meaning of this formula is that FirstClass commits to make the formula `int0` true, however under the condition that `pre` is true first. Hence, to invoke the service FirstClass, the formula `pre` must be satisfied.

Agent01 puts `info` into its `knowledge` of its `state`, and moves the formula `int0` from `plan` to `workflow` of its `state`. On the basis of the task formula, the agent01 knows that the formula `pre` describes the initial resource. The formula `pre` becomes the next intention of agent01, i.e., it is put into its `plan`.

Agent01 is looking for a service that can realize its current intention, i.e., satisfy the formula `pre`. From its `knowledge` the agent01 "knows" that TM-01 can realize this intention, see the formula (4.1). Agent01 sends a message with `protocolOrder` 001, and with `info` (created by agent01) having the formula:

```
    ( pre implies intentions( agent01 ) )
```

to the service TM-01.

The service TM-01 receives the message and displays the offers specified in the formula `pre` to the client. Then, TM-01 commits to realize this intention, creates appropriate element `commitment` in its `state`, i.e., the formula `( true )` is put into the element `formIn` of the `commitment`, whereas the formula `pre` is put into the element `formOut` of the `commitment`. Then the service TM-01 sends the appropriate message to the agent01, i.e., TM-01 replies to agent01 with the message with `protocolOrder` 021 having `info` (created by TM-01) that contains the following formula:

```
  ( true implies formInCommitment( TM-01 ) )
  and
  ( formOutCommitment( TM-01 ) implies pre )
```

Agent01 puts the `info` into its `knowledge`, and moves the formula `pre` from `plan` to `workflow` of its `state`. Since the precondition of the commitment is true, the element `plan` of the agent01's `state` becomes empty so that a workflow for realizing the task is already constructed. The workflow consists of the three services: TM-01, FirstClass, and TM-00. This workflow is represented by the formulas that are in the element `workflow` of agent's `state`. The formulas describe the commitments of the services involved into the workflow. The service TM-01 has committed to deliver a resource of type `Order`, satisfying the specification expressed in formula (5.2), to the service FirstClass. The service FirstClass has committed to deliver a resource of type `Invoice`, satisfying the specification expressed in formula (4.2), to the service TM-00.

Note that according to the protocol entish 1.0 (specified formally in Chapter 6) a service engaged in the workflow may cancel its commitment by sending a message with `protocolOrder` 002 to the agent. Once the service does so, the `state` of this service as well as the agent01's `state` must be changed. In the case of the service's `state` it is relatively simple, i.e., the appropriate element `commitment` is removed from `listOfCommitments` of this `state`. However, in the case of agent's `state` it is more complex, see details in the Chapter 6, Section 6.7. Analogously an agent may cancel service's commitment by sending a message with `protocolOrder` 020 to this service.

During the phase of workflow formation a service engaged in the workflow reserves temporarily some of its capabilities specified in the precondition of the commitment it has made. In the case of FirstClass, it reserves three places in the flights specified in the precondition, however, only by the time specified in the timeout. It is important that this very timeout is determined by the FirstClass itself.

4.6 Workflow execution

The workflow can be executed if agent01 sends synchronously the message of `protocolOrder` 222 to all the services engaged in the workflow. The `protocolSession` is set as the agent01's address, whereas the `info` in the message is created by agent01 and contains the formula `( true )`. The intended meaning of this message is that the workflow for realizing the agent01's task is already constructed and approved by the client, so that it may be executed. It is the signal for the services that are supposed to produce the initial resources to do so, and then to send them to the next services in the workflow.

Suppose that the client chooses one of the options (it is the one identified by the index "0.2") and creates a resource of type `Order` satisfying the constrains associated to this option.

Then, TM-01 puts this resource (as a file) into a www server. An element `constant` of type `Order` is created by service TM-01. The URL address of this file is put into the element `longName` of `constantName` of the `constant`. Let `orderURL` denote this `constant`.

Then, TM-01 sends a message with `protocolOrder` 321 to FirstClass, with `info` (created by TM-01) having the formula:

```
(
  ?order = orderURL
  and
  token( ?order ) = tok2
)
```

Once FirstClass receives the message, it knows (by the token `tok2` of the resource specified in this formula) what this resource is and what commitment is associated to this resource. Note that this very FirstClass has determined this token. So that FirstClass downloads the file that contains the resource of type `Order` produced by the client via TM-01, and sends confirmation to TM-01, i.e., a message with `protocolOrder` 331 with the `info` (created by FirstClass) having the following formula:

```
(
  isIn( ?order, FirstClass )
  and
  token( ?order ) = tok2
)
```

Once TM-01 receives this message, it sends a message with `protocolOrder` 333 to the agent01. The `info` of this message contains the formula `pre`, see the formula (5.2). This is the confirmation sent to the agent0 by the service TM-01, that the commitment of service TM-01 has been realized. The service TM-01 may put the `info` into its `knowledge`. Then, TM-01 removes the corresponding element `commitment` from `listOfCommitments` of its `state`.

At this very moment the FirstClass cancels the rest of reservations specified in the precondition of its commitment except the one chosen by the client in the order.

Then agent01, after receiving the message with `protocolOrder` 333, moves the intention `pre` from the element `workflow` to the element `realized` of its `state`.

Once FirstClass has got the resource specified in the formula `pre` of its commitment, it produces a resource of type `Invoice`, and puts it (as a file) on a www server. An element `constant` of type `Invoice` is created by the service FirstClass. The URL address of this file is put into the element `longName` of `constantName` of the `constant`. Let `invoiceURL` denote this `constant`.

Then, FirstClass sends a message with `protocolOrder` 321 to TM-00 with `info` (created by FirstClass) with the formula:

```
(
  ?invoice = invoiceURL
  and
  token( ?invoice ) = tok1
)
```

Once the service TM-00 receives the message, it downloads the invoice, and replies to FirstClass with a message with `protocolOrder` 331 having `info` (created by TM-00) that contains the following formula:

```
(
  isIn( ?invoice, TM-00 )
  and
  token( ?invoice ) = tok1
)
```

After receiving this message, FirstClass removes the commitment, puts the `info` into its `knowledge`, and sends a message with `protocolOrder` 333 to the agent01. The `info` of this message is created by the FirstClass, and contains the formula `int0`, see the formula (4.2). This is the confirmation sent to the agent 01 by the service FirstClass.

Then, agent01 moves the formula `int0` from `workflow` to `realized` of its `state`. Now, the `workflow` of agent01's `state` is empty. It means that the workflow has been executed successfully. The required invoice is in service TM-00 associated with the TaskManager of the client.

4.7 Transactional semantics

It may happen that during the phase of workflow execution one or more of the services fails to realize the commitments. The reason may be that the host (running the applications associated with the service) is down or the network is broken, or for some other reasons the service cannot realize its commitment; it means that the service cannot send the confirmation message with `protocolOrder` 333. In this case the service may send a message with `protocolOrder` 009 to agent01. The `protocolSession` of the message is set as the agent01's address, whereas the `info` is created by the service and contains the formula `( false)`. If, for some reasons, the service cannot send this message with `protocolOrder` 009, then after the timeout set for the commitment, the whole distributed transaction associated with this workflow will be canceled. This is because the final approval of this distributed transaction is done by the agent01 by sending synchronously the final confirmation to all the services participating in the workflow execution. In order to do so, agent01 must receive the confirmations (i.e., messages of `protocolOrder` 333) from all services involved in the workflow. The final approval consists in sending synchronously the message with `protocolOrder` 999 with `protocolSession` set as the agent01's address to all the services engaged in the workflow. The `info` of the message is created by agent01 and contains the formula `( true )`. This completes the distributed transaction.

However, the agent might not send the final approval after receiving all the confirmation from the services. If (for some reasons) the agent does not send the final approval by the timeout determined for the task realization, the distributed transaction is canceled automatically after this timeout. Note that the timeout for the task realization is encoded in agent's address, (see Chapter 6, Section 6.1) and this very address is put into the element `protocolSession` of every message sent during workflow construction phase as well as during workflow execution phase.

The agent01 is allowed to cancel the distributed transaction (after receiving all the confirmation from the services) by sending a message with `protocolOrder` 090. The `protocolSession` is set as the agent01's address, whereas the `info` of the message is created by the agent and contains the formula `( false )`.

If the transaction is canceled by the agent or the main timeout for task realization is over, every service participating in the workflow execution must undo the temporary results of performing its operation. Hence, before receiving the final confirmation from the agent approving the whole distributed transaction, the effects of performing its operation by a service are not persistent. So that after sending the invoice to the client, FirstClass has reserved only temporarily one seat in the flight specified in the invoice. It means that the appropriate ticket was not printed out and sent to the client. If the final confirmation is not delivered to the FirstClass before the timeout set for the task, the reservation will be canceled.

So that, even if the workflow was executed sucessfully, the client still has a chance to change his/her mind and cancel the whole transaction. Invoice is only a digital document and before approving the distributed transaction, it cannot influence the real world.

Once the final confirmation is received by FirstClass, the effect on the real world takes place, i.e., an appropriate ticket (for the flight specified in the order and in the invoice) is printed out and delivered to the client.

Chapter 5

Implementation of the Second Running Example

Let us consider again the second running example from Chapter 2, Section 2.3.2. The example is about buying books. This time we are going to present details of how the example may be implemented. The example serves for explaining the protocol entish 1.0 (specified formally in Chapter 6), i.e., how services are arranged into a workflow, how the workflow is executed, and finally how transactional semantics is implemented. In this example we also explain what indexes are and what they are for.

5.1 Service operation and state

To implement the example we must specify the resource types and the functions that are involved in the example.

First of all we must look at the contents of EntishDictionary that is the central repository (implemented in the enTish prototype, see Chapter 7) of instances of the schema `definitions.xsd`. We must find out if the names for types of resources, names of their attributes and the names for functions that the services BigBank and AnyBook have implemented as their operations have been already defined there. If they have not been defined, then we must define them either using EntishDictionary or manually creating a XML-document being an instance of `definitions.xsd`.

For the realization of the service AnyBook, we need the following resource types:

- `BookOrder`, that defines book orders. Let `?bookorder` be a variable of type `BookOrder`. This type may have several attributes, however we are interested in the following ones:
  - `Author( ?bookorder )`, returns string that specifies the author of the book.
  - `Title( ?bookborder )`, returns string that specifies the title of the book.
  - `Price( ?bookorder )`, returns string that specifies price.
  - `Person( ?bookorder )`, returns string that specifies details of the person who orders the book.
- `BookInvoice`, that defines book invoices. Let `?bookinvoice` be a variable of type `Invoice`. This type has the following attributes:
  - `author( ?bookinvoice )`, returns string that specifies the author of the book.
  - `title( ?bookinvoice )`, returns string that specifies the title of the book.
  - `price( ?bookinvoice )`, returns string that specifies price.
  - `person( ?bookinvoice )`, returns string that specifies details of the person who orders the book.
  - `Seller( ?bookinvoice )`, returns string that specifies details of the store that sells the book.

- `PayOrder`, that defines payment order. Let `?payorder` be a variable of type `PayOrder`. This type has the following attributes:
  - `amount( ?payorder )`, returns string that specifies the amount of money to be transferred.
  - `beneficiary( ?payorder )`, returns string that specifies the account number of the beneficiary of the money transfer.
  - `Person( ?payorder )`, returns string that specifies the account number of the person who makes the payment.
  - `subject( ?payorder )`, returns string that specifies the subject of the payment.
- `PayConfirm`, that defines payment confirmation. Let `?payconfirm` be a variable of type `PayConfirm`. This type has the following attributes:
  - `Amount( ?payconfirm )`, returns string that specifies the amount of money to be transferred.
  - `Beneficiary( ?payconfirm )`, returns string that specifies the details of the beneficiary of the money transfer.
  - `Subject( ?payconfirm )`, returns string that specifies the subject of the payment.

We define the following function:

```
SELL-BOOK: BookOrder X PayConfirm -> BookInvoice
```

that is, given a book order and appropriate payment confirmation, it produces an invoice for the book specified in the order.

We define also the following function:

```
PAYMENT: PayOrder -> PayConfirm
```

that is, given a payment order, it produces payment confirmation.

So that purchasing of a book by a client may be described in an abstract way as composition of these two functions, i.e.,

```
SELL-BOOK( ?bookorder, PAYMENT( ?payconfirm )).
```

Suppose that there is a service called AnyBook which is a bookstore. The service implemented the function `SELL-BOOK` as its operation. The service AnyBook has its `state`. The formula

```
isIn( ?bookorder, AnyBook ) and isIn( ?payconfirm, AnyBook )
```

is put into element `formIn` of `goal` of the `state` of the service. The formula

```
( isIn( ?bookinvoice, ?anyService )
  and
  ?bookinvoice = SELL-BOOK( ?bookorder, ?payconfirm ) )
```

is put into element `formOut` of `goal` of the `state` of the service. Once these two formulas are put into `formIn` and `formOut` of `goal` of the `state` of AnyBook, they describe the type of operation performed by the service AnyBook. That is, if a resource of type `BookOrder` and an appropriate resource of type `PayConfirm` are delivered to AnyBook, then it produces appropriate resource of type `BookInvoice` that can be delivered to any service (place) by AnyBook.

Suppose that there is a service called BigBank which is a service (a bank) that realizes payments. The service implements the function `PAYMENT`. The service BigBank has its `state`. The formula

```
  isIn( ?payorder, BigBank )
```

is put into element `formIn` of `goal` of the `state` of the service. The formula

```
 ( isIn( ?payconfirm, ?anyService )
   and
   ?payconfirm = PAYMENT( ?payorder ) )
```

is put into element `formOut` of `goal` of the `state` of the service. Once these two formulas are put into `formIn` and `formOut` of `goal` of the `state` of BigBank, they describe the type of operation performed by the service BigBank.

5.2 Publication of the operation type of AnyBook and BigBank

If a service wants to publish its operation type to an infoService, then it sends `message` of `protocolOrder` 000, and `protocolSession` set as the AnyBook's address. The `info` is created by the service, so that the address of AnyBook is put into the element `place` of the `info`. The element `formula` of the `info` contains the following formula:

```
 ( ( isIn( ?bookorder, AnyBook ) and isIn( ?payconfirm, AnyBook ) )
   implies formInOperationType( AnyBook )
 )
 and
 ( formOutOperationType( AnyBook ) implies
   ( isIn( ?bookinvoice, ?anyService )
     and
     ?bookinvoice = SELL-BOOK( ?bookorder, ?payconfirm )
   )
 )
```

Note that this single formula expresses the operation type of the service AnyBook. If the infoService agrees to publish the operation type of AnyBook, then the infoService puts the `info` into its `knowledge` of its `state`.

Then, infoService replies with the `message` of `protocolOrder` 111, with the same `protocolSession`, i.e., the AnyBook's address. The `info` of this message is created by infoService, that is, the address of the infoService is put into the element `place` of the `info`. The `info` contains the following formula:

```
   timeout(t1)
```

that sets the timeout for the validity of the entry (i.e., the `info` sent by AnyBook) in the infoService registry.

In the analogous way the service BigBank publishes its operation type to an infoService. It sends `message` of `protocolOrder` 000, and `protocolSession` set as BigBank's address. The `info` is created by the service, so that the address of BigBank is put into the element `place` of the `info`. The element `formula` of `info` contains the following formula:

```
( ( isIn( ?payorder, BigBank ) implies
    formInOperationType( BigBank )
  )
  and
  ( formOutOperationType( BigBank ) implies
    ( isIn( ?payconfirm, ?anyService ) and
      ?payconfirm = PAYMENT(?payorder)
    )
  )
)
```

Note that this single formula expresses the operation type of the service BigBank. If the infoService agrees to publish the operation type of BigBank, the infoService puts the `info` into its `knowledge` of its `state`.

Then, infoService replies with `message` of `protocolOrder` 111, with the same `protocolSession`, and with `info` (created by infoService) containing the formula:

```
    timeout(t2)
```

that sets the timeout for the validity of the entry (i.e., the `info` sent by BigBank) in the infoService registry.

5.3 Task

Client's task is expressed intuitively in the following way:
Purchase a book by J.R.R. Tolkien; timeout for task realizing: July 1, 2002.

The client is associated with a user interface called TaskManager. The client's task is expressed formally as the following formula:

```
(3.1)
    (  timeout("July 1, 2002")
       and
       isIn( ?bookinvoice, TM-00)
       and
       ?bookinvoice = SELL-BOOK( ?bookorder, PAYMENT( ?payorder ) )
       and
       token( ?bookinvoice ) = tok0
       and
       (
         index( ?bookinvoice ) = "0"
         and
         author( ?bookinvoice )="J.R.R. Tolkien"
```

```
        )
    )
```

The meaning of the formula is that the resource `?bookinvoice` is delivered to the service TM-00 (associated with the TaskManager) by July 1, 2002; the `?bookinvoice` is the output of some operation that implements the abstract function `SEL1-BOOK` that describes production of book invoice from book order and payment confirmation. The payment confirmation is the result of applying an operation that implements function abstract function `PAYMENT` to a pay order. The resource `?bookinvoice` is specified by exactly one attribute; it is `author`. The functions `index` and `token` are polymorphic functions defined for all resource types. They are introduced in `properEntish.xml`, see the Appendix. The role played by these functions in the composition protocol will be explained later on.

The TaskManager must create a service TM-00 for storing the final resource specified in the task formula. In order to do so the following formula:

```
(3.2)
      (  timeout("July 1, 2002")
         and
         isIn( ?bookinvoice, TM-00)
         and
         token( ?bookinvoice ) = tok0
         and
         (
          index( ?bookinvoice ) = "0"
          and
          author( ?bookinvoice )="J.R.R. Tolkien"
         )
       )
```

is put into the element `formIn` of a `commitment` of the `state` of the service TM-00.

The TaskManager also creates two services for delivering initial resources needed for the task realization. The name of the first one is TM-01. The formula `( true )` is put into the element `formIn` of a `goal` of the `state` of the service TM-01. Whereas the formula `isIn( ?bookorder, ?anyService )` is put to into the element `formOut` of the `goal` of the `state` of the service TM-01.

The name of the second service is TM-02. The formula `( true )` is put into the element `formIn` of a `goal` of the `state` of the service TM-02. Whereas the formula `isIn( ?payorder, ?anyService )` is put to into the element `formOut` of the `goal` of the `state` of the service TM-02.

These three auxiliary services are created dynamically by the TaskManager for a given task. The service TM-00 is supposed to receive and store the final resource specified in the task, i.e., the resource with the token `tok0`. The service TM-01 is supposed to deliver initial resource of type `BookOrder` whereas the service TM-02 to deliver an initial resource of type `PayOrder`. These initial resource are supposed to be created by the client associated with the TaskManager.

5.4 Agent

The task formula (3.1) is delegated to the agentServer where a process is created that is responsible for realization of this task. The process is called agent01 and has its own `state`. The task formula (3.1) is put into the element `formOut` of `goal` of the `state` of agent01.

A new element `info` is created by TM-01. The `info` contains the formula that states that TM-01 produces resources of type `BookOrder`. It is the following formula:

```
(4.1)
      ( true implies formInOperationType( TM-01 ) )
      and
      ( formOutOperationType( TM-01 ) implies
        isIn( ?bookorder, ?anyService)
      )
```

The `info` is put into the element `knowledge` of the agent01's `state`.

Also another element `info` is created by TM-02. This `info` contains the formula that states that TM-02 produces resources of type `PayOrder`. It is the following formula:

```
(4.1')
      ( true implies formInOperationType( TM-02 ) )
      and
      ( formOutOperationType( TM-02 ) implies
        isIn( ?payorder, ?anyService)
      )
```

The `info` is put into the element `knowledge` of the agent01's `state`.

Agent01 starts its algorithm. Initially, `plan` of `intentions` of agent01's `state` is empty. Because in the task formula (3.1) the function composition consists of two functions, the following formula becomes the first intention of agent01.

```
(4.2)
      (  timeout("July 1, 2002")
         and
         isIn( ?bookinvoice, TM-00)
         and
         ?bookinvoice = SELL-BOOK( ?bookorder, ?payconfirm )
         and
         token( ?bookinvoice ) = tok0
         and
         (
          index( ?bookinvoice ) = "0"
          and
          author( ?bookinvoice )="J.R.R. Tolkien"
         )
       )
```

Let `int0` denote this formula. The formula `int0` is put as the first element of `plan` of `intentions` of agent01's `state`. The natural question is how agent01 gets to know that the function PAYMENT returns value of type PayConfirm, that is, that the variable ?payconfirm (in the formula above) is of type PayConfirm. In order to explain this we must go back to the definition of the formula in the language Entish. The names of functions (as well as of types and relations) are elements of XML type ConceptName introduced in the schema formula.xsd. Hence, the function name PAYMENT denotes (in our informal notation) an element of the type ConceptName. This element (as well as any other element of the type ConceptName) consists of two elements: The first one is shortName and is of type xsd:string (and is supposed to be this very string "PAYMENT"), whereas the second one is longName and is of type xsd:anyURI. This very longName contains the URL to the XML document where the function name was defined. This document is an instance of the schema definitions.xsd and contains the definition of the function denoted by PAYMENT. This very definition contains the signature (interface) of the function as well as reference to its meaning in the definiendum part of the definition.

5.5 Workflow construction

The protocolSession of all the messages described below is set as agent01's address. Message of protocolOrder 001, with the info (created by agent01) having the formula:

```
int0 implies intentions( agent01 )
```

is sent by agent01 to an infoService (suppose that there is at least one). The intended meaning of the formula is that satisfaction of the formula int0 is an intention of agent01.

Suppose that infoService replies by forwarding the info of the message with protocolOrder 000 sent previously by the service AnyBook. The message sent by infoService to agent01 has protocolOrder 111, and protocolSession set as the agent01's address. Once the message is delivered to the agent, agent01 puts the info into its knowledge. So that agent01 knows that (according to the infoService) the service AnyBook can realize its current intention. Hence, agent01 sends the message with protocolOrder 001 to AnyBook. The info of this message is the same as the one sent by agent01 to the infoService in the previous message.

Suppose that the service AnyBook commits to realize the int0, however, under the condition described by the following formula:

```
(5.2)
    (  timeout("June 30, 2002")
       and
       isIn( ?bookorder, AnyBook) and token( ?bookorder ) = tok01
       and
       isIn( ?payconfirm, AnyBook) and token( ?payconfirm ) = tok02
       and
       (
         ( index( ?bookorder ) = "0.0"
           and
           Author( ?bookorder )="J.R.R. Tolkien"
           and
```

```
          Title( ?bookorder )="The Lord of the Rings"
          and
          Price( ?bookorder )="70euro"
          and
          index( ?payconfirm ) = "0.0"
          and
          Amount( ?payconfirm )="70euro"
        )
        or
        ( index( ?bookorder ) = "0.1"
          and
          Author( ?bookorder )="J.R.R. Tolkien"
          and
          Title( ?bookorder )="The Hobbit"
          and
          Price( ?bookorder )="60euro"
          and
          index( ?payconfirm ) = "0.1"
          and
          Amount( ?payconfirm )="60euro"
        )
        or
        ( index( ?bookorder ) = "0.2"
          and
          Author( ?bookorder )="J.R.R. Tolkien"
          and
          Title( ?bookorder )="The Silmarillion"
          and
          Price( ?bookorder )="50euro"
          and
          index( ?payconfirm ) = "0.2"
          and
          Amount( ?payconfirm )="50euro"
        )
      )
    )
```

Let the formula (5.2) be denoted by `pre`. The meaning of `pre` is that `?bookorder` having token `tok01` and `?payconfirm` having token `tok02` are delivered to AnyBook by June 30, 2002, and attributes of the `?bookorder` and `?payconfirm` should be chosen according to the ones listed in this formula. Actually, this attribute listing represents the options. These options are identified by indexes. So that the first option has index "0.0", the second one has index "0.1", whereas the third one has index "0.2". Note that the `token` as well as `index` of the input resources `?payconfirm` and `?bookorder` are determined by the service AnyBook that made this commitment.

The `state` of AnyBook is changed. A new element `commitment` of `listOfCommitments` of its `state` is created, and the formula `int0` is put into the element `formOut` of the `commitment`, whereas the formula `pre` is put into the element `formIn` of the `commitment`.

The service AnyBook replies to agent01 with a message with `protocolOrder` 021 with `protocolSession` set as agent01's address and with `info` (created by AnyBook) having the formula:

```
  ( pre implies formInCommitment( AnyBook ) )
  and
  ( formOutCommitment( AnyBook ) implies int0 )
```

The meaning of this formula is that AnyBook commits to make the formula `int0` true, however under the condition that `pre` is true first. Hence, to invoke the service AnyBook, the formula `pre` must be satisfied.

Agent01 puts `info` into its `knowledge` of its `state`, and moves the formula `int0` from `plan` to `workflow` of its `state`. On the basis of the task formula, agent01 knows that the formula `pre` describes one of the initial resources of type `BookOrder`, and one resource of type `PayConfirm` that is a result of processing a resource of type `PayOrder` by the function `PAYMENT`. The formula `pre` is decomposed into two intentions in the following way. The first intention (denoted by `int01` and corresponging to the resource `?bookorder`) is expressed as the formula:

```
(5.3)
      (  timeout("June 30, 2002")
         and
         isIn( ?bookorder, AnyBook) and token( ?bookorder ) = tok01
         and
         (
           ( index( ?bookorder ) = "0.0"
             and
             Author( ?bookorder )="J.R.R. Tolkien"
             and
             Title( ?bookorder )="The Lord of the Rings"
             and
             Price( ?bookorder )="70euro"
           )
           or
           ( index( ?bookorder ) = "0.1"
             and
             Author( ?bookorder )="J.R.R. Tolkien"
             and
             Title( ?bookorder )="The Hobbit"
             and
             Price( ?bookorder )="60euro"
           )
           or
           ( index( ?bookorder ) = "0.2"
             and
             Author( ?bookorder )="J.R.R. Tolkien"
             and
             Title( ?bookorder )="The Silmarillion"
             and
             Price( ?bookorder )="50euro"
           )
         )
       )
```

This formula becomes the next intention of agent01, i.e., it is put into its `plan`. The second intention (denoted by `int02` and corresponding to the resource `?payconfirm`) is expressed as the formula:

```
(5.4)
    (  timeout("June 30, 2002")
       and
       ?payconfirm = PAYMENT( ?payorder )
       and
       isIn( ?payconfirm, AnyBook) and token( ?payconfirm ) = tok02
       and
       (
         ( index( ?payconfirm ) = "0.0"
           and
           Amount( ?payconfirm )="70euro"
         )
         or
         ( index( ?payconfirm ) = "0.1"
           and
           Amount( ?payconfirm )="60euro"
         )
         or
         ( index( ?payconfirm ) = "0.2"
           and
           Amount( ?payconfirm )="50euro"
         )
       )
     )
```

This formula also becomes the next intention of agent01, i.e., it is also put into its `plan`.

Agent01 is looking for a service that can realize its first intention `int01`. From its `knowledge` the agent01 knows that TM-01 can realize this intention, see formula (4.1). Agent01 sends a message with `protocolOrder` 001, and with `info` (created by agent01) having the formula:

```
    int01 implies intentions( agent01 )
```

to the service TM-01.

The service TM-01 receives the message and displays the offers specified in the formula `int01` to the client. Then, TM-01 commits to realize this intention, creates appropriate element `commitment` in its `state`, i.e., the formula `( true )` is put into the element `formIn` of the `commitment`, whereas the formula `int01` is put into the element `formOut` of the `commitment`. Then the service TM-01 sends the appropriate message to agent01, i.e., TM-01 replies to agent01 with a message with `protocolOrder` 021 having `info` (created by TM-01) that contains the following formula:

```
( true implies formInCommitment( TM-01 ) )
and
( formOutCommitment( TM-01 ) implies int01 )
```

Agent01 puts the `info` into its `knowledge`, and moves the formula `int01` from `plan` to `workflow` of its `state`. Since the precondition of the commitment is true, no new intention is created by the agent.

Then, the agent looks for a service that can realize its second intention `int02`. A message of `protocolOrder` 001, with the `info` (created by agent01) containing the formula:

```
int02 implies intentions( agent01 )
```

is sent by agent01 to an infoService. Suppose that infoService replies by forwarding to agent01 the `info` of the message with `protocolOrder` 000 sent previously by the service BigBank to the infoService.

Then, agent01 puts the `info` into its `knowledge` and sends a message with `protocolOrder` 001 to the BigBank. The message contains the same `info` that was sent previously by the agent to the infoService.

Suppose that the service BigBank commits to realize the `int02`, however, under the condition described by the following formula:

```
(5.5)
      (  timeout("June 29, 2002")
         and
         isIn( ?payorder, BigBank) and token( ?payorder ) = tok021
         and
         (
           ( index( ?payconfirm ) = "0.0.0"
             and
             Amount( ?payconfirm )="70euro"
           )
           or
           ( index( ?payconfirm ) = "0.1.0"
             and
             Amount( ?payconfirm )="60euro"
           )
           or
           ( index( ?payconfirm ) = "0.2.0"
             and
             Amount( ?payconfirm )="50euro"
           )
         )
       )
```

Let the formula (5.5) be denoted by `pre021`. The meaning of `pre021` is that `?payorder` having token `tok021` is delivered to BigBank by June 29, 2002, and attributes of the `?payorder` should be chosen according to the ones listed in this formula. Actually, this attribute listing represents the options. These options are identified by indexes, so that the first option has index "0.0.0", the second one has index "0.1.0", whereas the third one has index "0.2.0". It is important to note that the service BigBank did not generate additional options, it means that in all these options the last number (from the left) is 0.

The `state` of BigBank is changed. A new element `commitment` of `listOfCommitments` of its `state` is created, and the formula `int02` is put into the element `formOut` of the `commitment`, whereas the formula `pre021` is put to the element `formIn` of the `commitment`.

The service BigBank replies to agent01 with a message with `protocolOrder` 021 with `protocolSession` set as agent01's address and with `info` (created by BigBank) having the formula:

```
( pre021 implies formInCommitment( BigBank ) )
and
( formOutCommitment( BigBank ) implies int02 )
```

The meaning of this formula is that BigBank commits to make the formula `int02` true, however under the condition that `pre021` is true. Hence, to invoke the service BigBank, the formula `pre021` must be satisfied.

Agent01 puts the `info` into the element `knowledge` of its `state`, and moves the formula `int02` from `plan` to `workflow` of its `state`. On the basis of the task formula, agent01 knows that the formula `pre021` describes the initial resource of type `PayOrder`. Hence, the formula `pre021` becomes its final intention and is put to `plan` of its `state`.

Now agent01 is looking for a service that can realize its final intention. From its `knowledge` the agent01 knows that TM-02 can realize this intention, see the formula (4.1'). Agent01 sends a message with `protocolOrder` 001, and with `info` (created by agent01) with the formula:

```
  pre021 implies intentions( agent01 )
```

to the service TM-02.

The service TM-02 receives the message and displays the options specified in the formula `pre021` to the client. Then, TM-02 commits to realize this intention, creates appropriate element `commitment` in its `state`, i.e., the formula `( true )` is put into the element `formIn` of the `commitment`, whereas the formula `pre021` is put into the element `formOut` of the `commitment`. Then the service TM-02 sends appropriate message to agent01, i.e., TM-02 replies to agent01 with the message with `protocolOrder` 021 having `info` (created by TM-02) that contains the following formula:

```
( true implies formInCommitment( TM-02 ) )
and
( formOutCommitment( TM-02 ) implies pre021 )
```

Agent01 puts the `info` into its `knowledge`, and moves the formula `pre021` from `plan` to `workflow` of its `state`. Since the precondition of the commitment is true, no new intention is created by the agent.

Now, `plan` of the agent01's `state` becomes empty so that a workflow for realizing the task has already been arranged. The workflow consists of the five services:

TM-00, AnyBook, BigBank, TM-01, and TM-02. This workflow is represented by the formulas that are in the element `workflow` of agent01's `state`. The formulas describe the commitments of the services involved into the workflow. The service TM-01 has committed to deliver a resource of type `BookOrder`, satisfying the specification expressed in formula (5.3), to the service AnyBook. The service TM-02 has committed to deliver a resource of type `PayOrder`, satisfying the specification expressed in the formula (5.5), to the service BigBank. The crucial point here is to grasp that the specifications (5.3) and (5.5) are interdependent. This interdependence is expressed by the indexes. It means that if the service TM-02 creates a resource with attribute `Amount` set as "70euro" (that corresponds to the option with index "0.0.0"), then the service TM-01 cannot create a resource with attribute `Title` set as "The Hobbit" that corresponds to the option with index "0.1" . Analogously, if the resource produced by TM-01 has attribute `Title` set as "Silmarillion" (corresponding to the option with index "0.2"), then TM-02 cannot produce a resource having attribute `Amount` set as "60euro" that corresponds to the option with index "0.1.0" . The reason is that there is a conflict of indexes that correspond to these options.

Note that the matching of the options is resolved at the level of indexes (identifying these options) without analyzing the attributes of the resources and semantic interrelations between them, e.g., without knowing that semantically the attribute `Price` of the resource of type `BookOrder` corresponds to the attribute `Amount` of the resource of type `PayOrder`.

Note that according to the protocol entish 1.0 (specified formally in Chapter 6) a service engaged in the workflow may cancel its commitment by sending a message with `protocolOrder` 002 to agent01. Once the service does so, the `state` of this service as well as the agent01's `state` must be changed. In the case of the service's `state` it is relatively simple, i.e., the appropriate element `commitment` is removed from `listOfCommitments` of this `state`. However, in the case of agent's `state` it is more complex, see details in Chapter 6, Section 6.7. Analogously the agent may cancel service's commitment by sending a message with `protocolOrder` 020 to this service.

During the phase of workflow formation a service engaged in the workflow reserves temporarily some of its capabilities specified in the precondition of the commitment it has made. In the case of AnyBook, it reserves three books from its stock specified in the precondition, however, only by the time specified in the timeout. It is important that this very timeout is determined by AnyBook itself.

5.6 Workflow execution

The workflow can be executed if the agent01 sends synchronously the message of `protocolOrder` 222 to all the services engaged in the workflow. The `protocolSession` is set as the agent01's address, whereas the `info` in the message is created by agent01 and contains the formula `( true )`. The intended meaning of this message is that the workflow for realizing the agent01's task is already constructed and approved by the client, so that it may be executed. It is the signal for the services that are supposed to produce the initial resources to do so, and then to send them to the next services in the workflow.

Suppose that the client chooses book entitled "The Hobbit" according to the options displayed by the TM-01 and TM-02. Then the client creates a resource of type `BookOrder` and a resource of type `PayOrder` satisfying the constrains associated to this option.

Then, TM-01 puts this resource of type `BookOrder` (as a file) into a www server. An element `constant` of type `BookOrder` is created by service TM-01. The URL address of this file is put into the element `longName` of `constantName` of the `constant`. Let `bookorderURL` denote this `constant`.

Then, TM-01 sends a message with `protocolOrder` 321 to AnyBook, with `info` (created by TM-01) having the formula:

```
(
  ?bookorder = bookorderURL
  and
  token( ?bookorder ) = tok01
)
```

Once AnyBook has received the message, it knows (by the token `tok01` of the resource specified in this formula) what this resource is and what commitment is associated to this resource. Note that this very AnyBook has determined this token. So that AnyBook downloads the file that contains the resource of type `BookOrder` produced by the client via TM-01, and sends confirmation to TM-01, i.e., a message with `protocolOrder` 331 with the `info` (created by AnyBook) having the following formula:

```
(
  isIn( ?bookorder, AnyBook )
  and
  token( ?bookorder ) = tok01
)
```

Once TM-01 has received this message, it sends a message with `protocolOrder` 333 to agent01. The `info` of this message contains the formula `int01`, see the formula (5.3). This is the confirmation that the commitment of service TM-01 has been realized. The service TM-01 may put the `info` into its `knowledge`. Then, TM-01 removes the corresponding element `commitment` from `listOfCommitments` of its `state`.

At this very moment AnyBook can cancel the rest of reservations specified in the precondition of its commitment except the one chosen by the client in the book order.

Agent01, after receiving the message with `protocolOrder` 333, moves the intention `int01` from the element `workflow` to the `realized` of its `state`.

Hence, AnyBook has got the first input resource that was specified in the precondition formula of its commitment.

Now, it is the turn of the service TM-02. It puts this resource of type `PayOrder` (as a file) into a www server. An element `constant` of type `PayOrder` is created by service TM-01. The URL address of this file is put into the element `longName` of `constantName` of the `constant`. Let `payorderURL` denote this `constant`.

Then, TM-02 sends a message with `protocolOrder` 321 to BigBank, with `info` (created by TM-02) having the formula:

```
(
    ?payorder = payorderURL
    and
    token( ?payorder ) = tok021
  )
```

Once BigBank has received the message, it knows (by the token of the resource specified in this formula i.e., `tok021`) what this resource is and what commitment is associated to this resource. Note that this very BigBank has determined this token. So that BigBank downloads the file that contains the resource of type `PayOrder` produced by the client via TM-02 and sends confirmation to TM-02, i.e., a message with `protocolOrder` 331 with the `info` (created by BigBank) having the following formula:

```
   (
     isIn( ?payorder, BigBank )
     and
     token( ?payorder ) = tok021
   )
```

Once TM-02 has received this message, it sends a message with `protocolOrder` 333 to the agent01. The `info` of this message contains the formula `pre021`, see the formula (5.5). This is the confirmation that the commitment of service TM-02 has been realized. The service TM-02 may put the `info` into its `knowledge`. Then, TM-02 removes the corresponding element `commitment` from `listOfCommitments` of its `state`.

Agent01, after receiving the message with `protocolOrder` 333, moves the intention `pre021` from the element `workflow` to the `realized` of its `state`.

BigBank has got the input resource that was specified in the precondition formula of its commitment. Hence, it produces a resource of type `PayConfirm`, and puts it (as a file) on a www server. An element `constant` of type `PayConfirm` is created by the service BigBank. The URL address of this file is put into the element `longName` of `constantName` of the `constant`. Let `payconfirmURL` denote this `constant`.

Then, BigBank sends a message with `protocolOrder` 321 to AnyBook, with `info` (created by BigBank) having the formula:

```
  (
    ?payconfirm = payconfirmURL
    and
    token( ?payconfirm ) = tok02
  )
```

Once AnyBook has received the message, it knows (by the token of the resource specified in this formula i.e., `tok02`) what this resource is and what commitment is associated to this resource. Note that this very AnyBook has determined this token. So that AnyBook downloads the file that contains the resource of type `PayConfirm` produced by the BigBank and sends confirmation to BigBank, i.e., a message with `protocolOrder` 331 with the `info` (created by AnyBook) having the following formula:

```
   (
    isIn( ?payconfirm, AnyBook )
    and
    token( ?payconfirm ) = tok02
  )
```

Once BigBank has received this message, it sends a message with `protocolOrder` 333 to agent01. The `info` of this message contains the formula `int02`, see the formula (5.4). This is the confirmation that the commitment of service BigBank has been realized. The service BigBank may put the `info` into its `knowledge`. Then, BigBank removes the corresponding element `commitment` from `listOfCommitments` of its `state`.

Agent01, after receiving the message with `protocolOrder` 333, moves the intention `int02` from the element `workflow` to the `realized` of its `state`.

Hence, AnyBook has got the second input resource that was specified in the precondition formula of its commitment.

Since AnyBook has all input resources, it produces a resource of type `BookInvoice`, and puts it (as a file) on a www server. An element `constant` of type `BookInvoice` is created by the service AnyBook. The URL address of this file is put into the element `longName` of `constantName` of the `constant`. Let `bookinvoiceURL` denote this `constant`.

Then, AnyBook sends a message with `protocolOrder` 321 to TM-00 with `info` (created by AnyBook) with the formula:

```
 (
   ?bookinvoice = bookinvoiceURL
   and
   token( ?bookinvoice ) = tok0
 )
```

Once the service TM-00 has received the message, it downloads the invoice, and replies to AnyBook with the message with `protocolOrder` 331 having `info` (created by TM-00) that contains the following formula:

```
  (
    isIn( ?bookinvoice, TM-00 )
    and
    token( ?bookinvoice ) = tok0
  )
```

After receiving this message, AnyBook removes the commitment, puts the `info` into its `knowledge`, and sends a message with `protocolOrder` 333 to agent01. The `info` of this message is created by AnyBook, and contains the formula `int0`, see the formula (4.2). This is the confirmation that the commitment of service AnyBook has been realized.

Then, agent01 moves the formula `int0` from `workflow` to `realized` of its `state`. Now, the element `workflow` of agent01's `state` is empty. It means that the workflow has been executed successfully. The required invoice is in service TM-00 associated with the TaskManager of the client.

Now, agent01 may approve the distributed transaction by sending the final confirmation to all the services involved in the workflow. If the agent decides to complete the trasnsaction, it sends synchronously a message with `protocolOrder`

999 with `protocolSession` set as the agent01's address to all the services engaged in the workflow. The `info` of the message is created by agent01 and contains formula `( true )`. This completes the distributed transaction.

Once the messages with final confirmation have been received by BigBank and AnyBook, the effect on the real world takes place, i.e., the appropriate money transfer is realized, and the book is delivered to the client.

Another aspects of transactional semantics, implemented in the protocol entish 1.0, are discussed at the end of the first example in Chapter 4, Section 4.7.

Chapter 6

Service Composition Protocol entish 1.0

6.1 Introduction

The protocol entish 1.0 is a specification of conversation between an agent, services, and an infoService. They exchange messages with specific contents according to some order.

Sending a message as well as receiving one may cause a change in the sender's and recipient's `state`. A `message` of specific type (`protocolOrder` in our convention) with a specific contents can be sent if the sender's `state` satisfies certain condition associated with this `message` type.

Note, that the data structures: `message`, `state`, and `info` are XML documents created according to the schemes defined in `message.xsd`, `state.xsd`, and `info.xsd`, see the Appendix. The most important elements of each of such structures are Entish formulas. Entish is a language for service description. Its syntax is defined in `formula.xsd` and `properEntish.xml`, so that Entish formulas are XML-elements. Entish is an open language. Names for new concepts can be introduced according to the schema `definitions.xsd` in a distributed way, i.e., anyone can introduce any concept or a collection of concepts that may be regarded as an ontology. Any name of type, function, and relation that occur in a well formed Entish formula must contain the URI of a XML document (being an instance of the `definitions.xsd` schema) where these names are introduced. The document must be available on www.

Since XML syntax of Entish is hard to read, for the purpose of the presentation we have also introduced the translation of the XML-syntax into a typical syntax of a language of first order logic. See the Chapter 3, Section 3.3 for details. In this protocol specification Entish formulas will be represented in this typical syntax. So that, if `psi` and `phi` are formulas, then `(psi or phi)`, `(psi and phi)`, `(psi implies phi)` are formulas. Terms and atomic formulas are defined in the usual way however, quantifiers and explicit negation are not used.

Let us start with the specification of the elements of `message`. The element `to` and the element `from` of the `header` of the `message` are respectively the recipient address and the sender address. They are addresses of agents, services, and infoServices. The format for the addresses is the following:

```
(1.1)
      http://host.name/path/party_id
```

where `party_id` is either a pointer to agent, service, or infoService. In the case of agent, the string `party_id` is of the form `agentname/date-time` where `date-time` is a string of the type `xsd:dateTime`, and denotes the timeout set for the task delegated to the agent for realization.

It is supposed that HTTP is the transport protocol for delivering (using the POST method) the messages between conversation parties.

The element `protocolName` of `message` is set as `entish`. The element `protocolVersion` is set as `1.0`.

The `message` contents, i.e., the `body` contains the element `listOfInfos`. The element `listOfInfos` contains the elements `signedInfo`. The element `signedInfo` contains element `info` and the optional element `signature`. In the present version of the protocol it is assumed that exactly one element `signedInfo` is in the `listOfInfos`, and there is no element `signature` in the `signedInfo`. The element `info` contains three elements: `formula, time`, and `place`. It represents a fact, i.e., that the `formula` (i.e., formula of our language Entish defined in the schema `formula.xsd`) was true at the `time` and in the `place`.

Please note that the operators "`and`" , "`or`", and "`implies`" (defined in the `formula.xsd`) can have arbitrary (at least two) number of formulas as elements. Usually, these operators are binary. So that we introduce the following convention: The formulas:

```
( phi1 and phi2 and ...  phiN )
( phi1 or phi2 or ...  phiN )
```

represent (in our typical syntax) the appropriate XML-formulas with the number of elements equal to N.

From now on, by the name of function, or type, or relation we mean the the element `functionName`, or `typeName`, or `relationName` created in a XML document according to the schema `definitions.xsd`.

In the document `properEntish.xml` (that is an instance of the schema `definitions.xsd`) we have introduced several primitive names for types, relations, and functions. The types `Service` and `Agent` are special ones; they describe the processes associated with agents and services. Their elements are called agents and services. There are also auxiliary types: `Time, Index, Token`. Elements of these types are special strings and correspond to the type `xsd:string` of XML Schema.

For any other type that can be introduced to Entish via `definitions.xsd`, its elements are called resources.

A resource or an agent, or a service as well as an element of any type is described in Entish by a term, i.e., by `variable`, `constant`, or by a complex term, see `formula.xsd` in Appendix and Chapter 3, Section 3.3 for details. However, in the case of agent and services, if it is a `constant`, i.e., fixed name of agent or service, then the element `longName` of the `constantName` of the `constant` is an agent's or service's address created according to the format defined in (1.1); whereas the element `shortName` contains `party_id`.

If the `constant` is the name of a resource, then the element `longName` of the `constantName` of the `constant` is an URL that points to the very place where the resource is stored. Hence, it is natural that resources are transported by the method GET of the HTTP protocol.

For the auxiliary types, i.e., `Time, Index, Token` whose elements are strings, a `constant` of any of these types contains this very string in the `shortName` of its element `constantName`, and there is no element `longName`.

For any message order, a canonical form of formula will be defined that occurs in its element `info`. The reason for introducing canonical formula format is, first of all, to exclude any ambiguity with their interpretation, and also to reduce reasoning overload.

6.2 Service state and operation type

Suppose that a service (having name `service_fun`) implements an abstract Entish function `fun`. The function has K parameters (input variables): `?input0, ?input1, ... , ?inputK`.

The service's address (defined according to (1.1)) is put in the element `owner` of `state`, The element `state` represents the state of the `service_fun` in the protocol entish 1.0.

Type of operation performed by service_fun consists of:

(i) precondition necessary for invoking the service:

```
(
 isIn( ?input0, service_fun )
 and
 isIn( ?input1, service_fun )
 and
 ...

 isIn( ?inputK, service_fun )
 and
 propIn( ?input0, ?input2, ?inputK )
)
```

This formula (denoted by `for_in`) is put into the element `formIn` of the `goal` of the `state` of the service_fun. Note that the formula

```
propIn( ?input0, ?input2, ?inputK )
```

is an arbitrary Entish formula specifying the input of the service.

(ii) post condition describing the result of operation performed by the service:

```
(
 isIn( ?finRes, ?anyService )
 and
 ?finRes = fun( ?input0, ?input1, ...  ?inputK )
 and
 propOut( ?finRes )
)
```

This formula (denoted by `for_out`) is put into the element `formOut` of `goal` of `state` of the service_fun. Note that `propOut( ?finRes )` is an arbitrary Entish formula specifying the output of the service.

6.3 Publication of the service operation type to an infoService

If the service wants to register to an infoService, then it sends a `message` of `protocolOrder` set as 000, and `protocolSession` set as `service_fun`'s address. The `info` is created by the service, so that the address of `service_fun` is put in the element `place` of the `info`. The element `formula` of `info` contains a formula of the form:

```
(2.1)
     (
       for_in implies formInOperationType( service_fun ) )
       and
       ( formOutOperationType( service_fun ) implies  for_out )
     )
```

The `info` is stored in infoService for some time specified by the formula `timeout( t1 )`. Note that `t1` is a `constant` of type `dateTime`. The infoService may reply with a `message` of `protocolOrder` set as 111, `protocolSession` set as `service_fun`'s address. The `info` in the `message` is created by the infoService, so that the address of infoService is put in the element `place` of the `info`. The element `formula` of `info` contains a formula of the form:

```
    timeout( t1 )
```

The intended meaning is that registration is valid as long as the formula `timeout( t1 )` is true. Once the service has received the `message`, the `info` of this `message` is put into `knowledge` element of the `state` of the service. Before the timeout is over, the service must repeat the registration procedure taking into account the current value of the GMT time, i.e., the service must update the value of the element `time` of the `info` in the registration message.

6.4 Task and agent state

Type of task is defined as a term of the Entish language; see `formula.xsd` in Appendix for the term definition. However, we consider only those terms that do not contain subterms of the type `Agent` or of the type `Service`.

For the simplicity of the presentation let us assume that the term in question is a composition of functions already defined in Entish, and there are no constants in the term, and all variables in the composition have different names, i.e., any variable occurs exactly once in the term. Once we grasp this simple case, it will be clear how to extend the protocol to the case where the type of task is an arbitrary term. The extension is done in Section 6.9.

The canonical form of the simple term defining task type is the following:

```
(3.1)
      fun( fun0(...), fun1(...), ... , funN(...) )
```

Note, that some of the subterms inside `fun( ... )` may be variables.
Let

```
(3.2)
      ?x0, ?x1, ... , ?xJ
```

be all the variables occurring in the term (3.1). It may happen that one (or more) of the terms `fun0(...), fun1(...), ... , funN(...)` is a variable listed above.

The notion of task is crucial in our approach. It is created by the client side, i.e., an application or by a user via a TaskManager. It is supposed that the application or the Task Manager is responsible to deliver initial resources for the task realization as well as for storing the final resource being the result of a successful realization of the task.

The canonical format of task formula is defined as follows:

```
(3.3)
     (
       timeout( date0 )
       and
       isIn( ?finRes, service0 )
       and
       ?finRes = fun( fun0(...), fun1(...), ... , funN(...) )
       and
       token( ?finRes ) = tok0
       and
       prop( ?finRes )
     )
```

where the format of the formula `prop( ?finRes )` is defined in (3.4), and service0 is the name of a service (created by an application or TaskManager) that is supposed to receive the final resource, i.e., the result of successful task realization. Note that `date0` is a `constant` of type `dateTime`, `service0` is a `constant` of type `Service` whereas `tok0` is a `constant` of type `Token`. The following formula

```
     (
       timeout( date0 )
       and
       isIn( ?finRes, service0 )
       and
       token( ?finRes ) = tok0
       and
       prop( ?finRes )
     )
```

is put into the element `formIn` of a `commitment` of `listOfCommitments` of the `state` of the service0. Whereas the formula `( true )` is put into the element `formOut`. Note, that service0 is regarded as an ordinary service so that it also exchanges messages according to the protocol entish 1.0, see Section 6.5 for more details concerning service commitments.

Note that in the formula (3.4) we apply the following conventions: `"0", "1", ... ,"K"` denote constants of type `Index`, where a string inside ”” is put into element `shortName` of `constantName` of the appropriate `constant`. The same convention is applied also to the constants of type `dateTime`, and `Token`.

The canonical format of the formula `prop( ?finRes )` in (3.3) is defined as follows:

```
(3.4)
     (
       ( index( ?finRes ) = "0" and prop0( ?finRes ) )
       or
       ( index( ?finRes ) = "1" and prop1( ?finRes ) )
       or
```

```
    ...
    or
    ( index( ?finRes ) = "K" and propK( ?finRes ) )
  )
```

Hence, the formula `prop( ?finRes )` is the disjunction of options. Each of the options is identified by an index. Usually, the formula specifing such option describes the final resource `?finRes` in terms of its attributes; see the examples in Chapters 4 and 5.

For every variable from the sequence (3.2), say `?xj`, let service_xj denote the name of a service (being a part of an application or created by TaskManager), that can deliver resources of the same type as the type of the variable `?xj`. These services are either associated (by an application) with the task, or dedicated by TaskManager to the task. Note, that these services also exchange messages according to the protocol entish 1.0, in the very same way as ordinary services do. Therefore, any of the services, say service_xj, has its own `state`. The formula `( true )` is put into the element `formIn` of `goal` of the `state` of the service service_xj, whereas the formula `( isIn( ?xj, ?anyService ) )` is put into the element `formOut` of the `goal` of the `state`.

Note, that the services: service0, and service_xj for j=0, 1, ... J, as well as the `state` of an agent process responsible for realization of the task, must be created by an application or a TaskManager.

Let agent0 be the name of an agent process. The agent0's address (defined according to (1.1)) is put into element `owner` of the `state` of the agent. Then, the task formula (3.3) is put into element `formOut` of the element `goal` of `state` of the agent. And for every j = 0, 1, 2, ... ,J; the following formula:

```
(3.5)
  (
    ( true implies formInOperationType( service_xj ) )
    and
    ( formOutOperationType( service_xj ) implies
      isIn( ?xj, ?anyService)
    )
  )
```

is put into the element `formula` of `info`. The address of the service_j is put into the element `place` of the `info`. These elements `info` are put into the element `knowledge` of the agent's `state`.

Note that the formula (3.5) has a similar form as the formula (2.1). Hence, it means that an agent knows that these services can deliver the initial resources (of the types specified by variable `?xj`) that are needed for task realization.

Once the agent's `state` is completed as it was described above, it represents the agent in the entish1.0 protocol. From the protocol's point of view, the way the `state` and the corresponding services were constructed is not important. They may be constructed by an application, TaskManager, or a user.

6.5 Intentions

Canonical format of the intention derived from the task formula (3.3) is defined as:

```
(4.1)
      (
        timeout( date0 )
        and
        isIn( ?finRes, service0 )
        and
        ?finRes = fun( ?res_0, ... , ?res_N )
        and
        token( ?finRes ) = tok0
        and
        prop( ?finRes )
      )
```

For every `j=0,1, ... , N` the variable `?res_j` corresponds to the term `funj(...)` in (3.1). Note that the type of variable `?res_j` is the the same as the type returned by the function `funj`. Let the formula (4.1) be denoted by `int`. It is put into the element `plan` of the element `intentions` of the `state` of agent0.

The agent may send a `message` of `protocolOrder` set as 001 and `protocolSession` set as agent0's address to an infoService or to a service. The `info` of `message` is created by the agent, i.e., the agent's address is put into the element `place` of the `info`. The canonical format of formula in the element `formula` of the `info` is of the following form:

```
(4.2)
        ( int implies intentions( agent0 ) )
```

If a `message` of `protocolOrder` set as 001 was sent to an infoService, then the infoService may reply with a `message` of `protocolOrder` set as 111 and `protocolSession` set as agent0's address. The `info` of the `message` is one of the elements `info` sent by services to the infoService; see the Section 6.2.1. It is supposed that it is the `info` created and sent by a service (to the infoService) that can realize the formula `int`, i.e., the agent's intention.

6.6 Commitments

Suppose that a service (its name is service1) receives the `message` of `protocolOrder` set as 001 from agent0. Then, service1 stores the `info` of the `message` in the element `knowledge` of its `state`. Suppose also that service1 agrees to realize the agent's intention. Then, service1 replies with a `message` of `protocolOrder` set as 021 and `protocolSession` set as agent0's address. The `info` of the `message` is created by service1, that is, the service1's address is put into the element `place` of the `info` of the `message`. Formula to be put into element `formula` of the `info` has the following canonical format:

```
(5.1)
```

```
(
 ( pre implies formInCommitment( service1 ) )
 and
 ( formOutCommitment( service1 ) implies int )
)
```

Once the `message` of `protocolOrder` set as 021 has been received by the agent, the agent puts the `info` of the `message` into its element `knowledge` of its `state`, and then moves the formula `int` from the element `plan` of `intentions` of its `state` into the element `workflow`. Then, agent0 is obliged to create next intentions from the formula `pre` and put them into the element `plan` of `intentions` of its `state`.

Once the `message` was sent successfully by the service1 to the agent0, a new element `commitment` of the element `listOfCommitments` of the `state` of service1 is created. Formula `pre` is put into the element `formIn` of the new `commitment` whereas the formula `int` is put into the element `formOut` of the new `commitment`.

The canonical format of the formula `pre` occurring in the formula above (5.1) is the following:

```
(5.2)
     ( timeout( date1 ) and
       isIn( ?res_0, service1 )  and  token( ?res_0 ) = tok00  and
       isIn( ?res_1, service1 )  and  token( ?res_1 ) = tok01  and
       ...                       and ...                       and
       isIn( ?res_N, service1 )  and  token( ?res_N ) = tok0N  and
       (
         option00( ?res_0, ?res_1, ...  , ?res_N ) or
         option01( ?res_0, ?res_1, ...  , ?res_N ) or
         ...                                       or
         option10( ?res_0, ?res_1, ...  , ?res_N ) or
         option11( ?res_0, ?res_1, ...  , ?res_N ) or
         ...                                       or
         option20( ?res_0, ?res_1, ...  , ?res_N ) or
         option21( ?res_0, ?res_1, ...  , ?res_N ) or
         ...                                       or
         optionK0( ?res_0, ?res_1, ...  , ?res_N ) or
         optionK1( ?res_0, ?res_1, ...  , ?res_N ) or
         ...
       )
     )
```

Where the number K is the number of options in the task and intention formulas, see (3.3), (3.4), and (4.1). The canonical format of the formula

```
    optionij( ?res_0, ?res_1, ...  , ?res_N )
```

(for `i=0, 1, ...  , K`; and `j=0,1,....` , where for each `i` the number of `optionij` is of course finite) is the following:

```
(
  index( ?res_0 ) = "i.j" and prop0ij( ?res_0 ) and
  index( ?res_1 ) = "i.j" and prop1ij( ?res_1 ) and
  ...                                           and
  index( ?res_N ) = "i.j" and propNij( ?res_N )
)
```

It is important to note, that the index `"i.j"` corresponds to this very option `optionij`. The option is the conjunctions of formulas. Any of the formulas corresponds to a separate variable; the formula determines the index for the variable, and describes it. Since it is a conjunction, for the option to be true all formulas must be true. Later on the formulas will be separated (see (5.4)) but the index will be attached to each one to indicate that they are bound by the conjunction.

Note that any option having index `"i.*"` is generated from the option having index `"i"`, see the formula (3.4), and the examples in Chapters 4 and 5.

Each of the formulas:

```
prop0ij( ?res_0 )
prop1ij( ?res_1 )
...
propNij( ?res_N )
```

is a conjunction of atomic formulas describing the resources `?res_0, ?res_1, ... ?res_N` in terms of their attributes.

For example, the formula

```
prop1ij( ?res_1 )
```

may be of the following form:

```
(
  atribute0( ?res_1 ) = "string0"
  and
  atribute1( ?res_1 ) = "string1"
  and
  atribute2( ?res_1 ) = "string2"
  and
  ...
)
```

where the functions `atribute0, atribute1, ...` are attributes of the same type as the type of the variable `?res_1`. Note, that according to our convention, attributes of a type (that is, functions defined on this very type) are usually defined in the very XML-document (an instance of `definitions.xsd`) where this type is defined.

Now, let's go back to agent0. It is obliged to create next intentions from the formula `pre`, i.e., (5.2). Each of these intentions is associated with one of the input resources specified in the formula `pre` and denoted by `?res_0, ?res_1, ... ?res_N`.

For example, for the input resource `?res_j` (that corresponds to the term `funj(...)`, see (3.1)) the intention is of the following form:

```
(5.3)
     (
       timeout( date1 )
       and
       isIn( ?res_j, service1 )
       and
       ?res_j = funj( ?res_j0, ?res_j1, ... )
       and
       token( ?res_j ) = tok0j
       and
       Prop_j( ?res_j )
     )
```

where `?res_j0, ?res_j1, ...` are the variables that ocurr in the term `funj(...)`, see (3.1). Let this formula be denoted by `intj` . Note, that the format of formula `intj` is the same as the format of the formula (4.1). Hence, the canonical format of intention formula is one and the same.

The formula `Prop_j( ?res_j )` in the formula (5.3) is defined as

```
(5.4)
     (
       ( index( ?res_j ) = "0.0" and propj00( ?res_j ) )
       or
       ( index( ?res_j ) = "0.1" and propj01( ?res_j ) )
       or
       ...
       ( index( ?res_j ) = "1.0" and propj10( ?res_j ) )
       or
       ( index( ?res_j ) = "1.1" and propj11( ?res_j ) )
       or
       ...
       ( index( ?res_j ) = "K.0" and propjK0( ?res_j ) )
       or
       ( index( ?res_j ) = "K.1" and propjK1( ?res_j ) )
       or
       ...
     )
```

Now, let us have a look at the formulas (3.3) and (5.3). If for some `i = 0,1,2, ... N`: the term `funi(...)` is one of the variables from the sequence (3.2), say `?x_j`, then the intention formula for the resource `?res_i` is defined as

```
(5.5)
     (
       timeout( date1 )
       and
       isIn( ?res_i, service1 )
       and
       ?res_i = ?x_j
       and
```

```
        token( ?res_i ) = tok0i
        and
        Prop_i( ?res_i )
      )
```

These new intentions (i.e., of the form defined in (5.3) and (5.5)) are put into the element `plan` of `intentions` of the `state` of agent0. Once the agent has received a `commitment` from a service to realize an intention, this intention is moved to the element `workflow` of the `intentions` of `state` of agent0. And so on.

Agent0 repeats this rutine until it reaches the services that deliver the initial resources (see 3.5), i.e., the services: service_xj that delivers resources of the same type as the type of the variable `?xj`, for `j=0,1, ..., J`. The variables occur in the task formula (3.2).

Now, suppose that agent0 sends intention `intK` to service_xj, and suppose that service_xj agrees to realize this intention. Then, the commitment formula of the service is the following:

```
  (
    ( true implies formInCommitment( service_xj ) )
    and
    ( formOutCommitment( service_xj ) implies intK )
   )
```

Then, agent0 moves `intK` from `plan` to `workflow`. Since the formula `pre` in the `commitment` is `( true )`, one chain of intentions was closed, i.e., agent0 has no new intention to put into `plan`. If the agent reaches all services (that are supposed to deliver the initial resources for task realization), then the element `plan` of `intentions` of the agent's `state` will be empty. It means that agent0 has already arranged services into a workflow that can realize its task.

The intentions formulas in the `workflow` form a tree order. The root of the tree is the intention formula (4.1) generated from the task formula (3.3). It has N children nodes corresponding to the terms

```
   fun0(...),  fun1(...), fun2(...), ... funN(...)
```

These nodes are of the form of formulas defined in (5.3) and (5.5). If a node is of the form (5.5), then it is a leaf node of the tree. If it is of the form (5.3), then it is a parent node and has other children. Hence, the tree corresponds to the function composition in the task formula (3.3).

Since for any intention formula from the element `workflow` there is a service that has committed to realize this intention, these services are composed into the workflow. The workflow also has a structure of the tree. It starts its execution from the leaf services that deliver the initial resources. The initial resources are delivered to the next services as input resources in the tree order. Each of the next services produces an output resource from the input resources and delivers the output to the next service in the tree order. And so on. Finally, the root service stores the final resource. If the final resource is delivered successfully, then it means that the task has been realized.

6.7 Control of workflow formation

Once the element `plan` of `intentions` of the `state` of agent0 is empty, agent0 may send (synchronously!) a `message` of `protocolOrder` set as 222 and of `protocolSession` set as agent0's address to all the services engaged in the workflow. The element `info` of the `message` is constructed by the agent so that the agent's address is put into the element `place` of `info`. The formula `( true )` is put into the element `formula` of the `info`.

Once a leaf service has received this message, it is obliged to start the workflow execution, see the next Section.

During workflow construction phase (i.e., before sending a `message` of `protocolOrder` set as 222 by the agent), the agent may cancel the `commitment` of a service as well as the service may also cancel its own `commitment`.

A service may cancel its `commitment` by sending a `message` of `protocolOrder` set as 002 and of `protocolSession` set as agent0's address to agent0. The `info` of the `message` is the same `info` as the one in the `message` of `protocolOrder` set as 021 and `protocolSession` set as agent0's address sent before to agent0 by the service. After sending this message, the corresponding element `commitment` is removed from the `listOfCommitments` of the `state` of the service.

Once the agent receives the `message` of `protocolOrder` 002 from the service, it is obliged to move back the corresponding intention formula (say `intK`) from `workflow` to `plan` of `intentions` of its `state`. Then, agent0 has to remove all intention formulas from the `workflow` that come later than `intK` according to the tree order, and to cancel all commitments associated with the removed intentions.

To cancel a `commitment` of a service, agent0 sends the `message` of `protocolOrder` set as 020 and of `protocolSession` set as agent0's address to the service. The `info` of the `message` is the same `info` as in the `message` of `protocolOrder` set as 021 sent previously by the service to agent0. Once the service receives this `message` of `protocolOrder` set as 020, it removes the corresponding element `commitment` from `listOfCommitments` of its `state`. If the message does not follow the `message` of `protocolOrder` set as 002, then, after sending the message, agent0 moves the associated intention back to the element `plan` of its `state`.

6.8 Workflow execution and control

Suppose that the element `plan` of `intentions` of the `state` of agent0 is empty, and agent0 has already sent `message` of `protocolOrder` set as 222 and of `protocolSession` set as agent0's address to all the services composed into the workflow. So that the workflow execution was started by agent0. Now, suppose that service (of name service2) has committed to realize one of the agent's intentions i.e., the previous elements of `plan` of agent0.

Suppose that this intention is one of the intentions of the form (5.3), i.e., it is the following formula:

```
(7.1)
    (
      timeout( date1 )
      and
      isIn( ?res_j, service1 )
      and
      ?res_j = funj( ?res_j0, ?res_j1, ... )
      and
      token( ?res_j ) = tok0j
      and
      Prop_j( ?res_j )
    )
```

Let this formula be denoted by `intj` . Once service2 has committed to realize this intention, the formula `intj` was put into the element `formOut` of the appropriate element `commitment` of `listOfCommitments` of the `state` of service2.

So that this formula was moved from `plan` of the `intentions` of the `state` of agent0, to the element `workflow`.

Now, suppose that the service2 has got all the initial resources it needed to produce its output resource. Service2 produced the resource specified in the formula `intj` , and stored it in its HTTP server giving it the URL. Let `URLj` be a `constant` of the same type as the type of the resource, i.e., the URL is put into the element `longName` of `constantName` of the `constant`.

Then, service2 sends the `message` of `protocolOrder` set as 321 and `protocolSession` is set as agent0's address to service1. The `info` of a `message` is created by service2, so that the address of service2 is put into element `place` of the `info`. The formula in the element `formula` of the `info` is of the following form:

```
  (
    ?res_j = URLj
    and
    token( ?res_j ) = tok0j
  )
```

Once service1 has received the `message` of `protocolOrder` set as 321, it downloads the resource from the URL. After successful downloading, the service1 replies with the `message` of `protocolOrder` set as 331 and `protocolSession` set as agent0's address to service2. The `info` of the `message` is created by service1, so that the address of service1 is put into the element `place` of the `info`. The formula in the element `formula` of the `info` is of the following form:

```
   (
     isIn( ?res_j, service1 )
     and
     token( ?res_j ) = tok0j
   )
```

Once service2 has received a `message` of `protocolOrder` set as 331, it puts the `info` of the `message` into the element `knowledge` of its `state`. Then, service2 sends a `message` of `protocolOrder` set as 333 and `protocolSession` set as agent0's address to agent0. The `info` of the `message` is created by service2, so that the address of service2 is put into the element `place` of the `info`. The formula in the element `formula` of the `info` is the formula from the element `formOut` of the appropriate `commitment` element of `listOfCommitments` of the `state` of service2. Note that this formula is the formula `intj`, i.e., (7.1). Once the `message` has been sent successfully, the `commitment` element is removed from the `listOfCommitments` of `state` of service2. Agent0 receives the `message` and moves the formula `intj` from element `workflow` of the `intentions` of its `state` to the element `realized` of the `intentions`.

It is possible that the timeout `timeout( date1 )` specified in the commitment formula (5.2) is over, and service1 did not receive all the messages of `protocolOrder` set as 321 associated with its input resources. Then, the appropriate element `commitment` is removed from the `listOfCommitments` of the `state` of service1. Then, service1 is obliged to send a message of `protocolOrder` set as 009 to agent0, see Section 6.10 for details.

6.9 Task formula is an arbitrary term

Suppose that one of the terms `fun0(...), fun1(...), ... , funN(...)` , say `funi(...)`, in (3.1) is a constant `const_i` , i.e., the element `longName` of the `constantName` of the constant contains the `URL_i` of a resource. Suppose that service1 replies with the commitment formula (5.2), so that the resource `?res_i` should be delivered to service1, and the token `tok0i` is assigned to this resource by service1.

Once the workflow execution is initiated by the agent, agent0 sends a `message` of `protocolOrder` set as 321 and `protocolSession` set as agent0's address to service1. The `info` of the `message` is created by agent0, so that the address of agent0 is put into the element `place` of the `info`. The formula in the element `formula` of the `info` is of the following form:

```
(
  ?res_i = const_i
  and
  token( ?res_i ) = tok0i
)
```

Once service1 has received the `message` of `protocolOrder` set as 321, it downloads the resource from `URL_i`. After successful downloading, service1 replies with a `message` of `protocolOrder` set as 331 and `protocolSession` set as the agent0's address to agent0. The `info` of the `message` is created by service1, so that the address of service1 is put into element `place` of the `info`. The formula in the element `formula` of the `info` is of the following form:

```
(
  isIn( ?res_i, service1 )
  and
  token( ?res_i ) = tok0i
)
```

Finally, agent0 stores this `info` in `knowledge` of its `state`.

Now, suppose that two of the terms `fun0(...)`, `fun1(...)`, `...` , `funN(...)` , in (3.2) are one and the same. Let `fun_i(...)` be the same as `fun_j(...)` . Suppose that service1 replies with the commitment formula (5.2), So that the resource `?res_i` should be delivered to service1, and the token `tok0i` is assigned to this resource by service1.

And the resource `?res_j` should be delivered to service1, and the token `tok0j` is assigned to this resource by service1.

Then, agent0 creates only one intention that has the following form:

```
(
  timeout( date1 )
  and
  isIn( ?Res, agent0 )
  and
  ?Res = funj( ?res_j0, ?res_j1, ... )
  and
  token( ?Res ) = Tok
  and
  Prop( ?Res )
)
```

Note that in this intention formula the resource `?Res` must be delivered to agent0, and the `constant Tok` is created by agent0.

The formula `Prop( ?Res )` is defined as

```
(
  ( index( ?Res ) = "0.0" and propj00( ?Res ) and propi00( ?Res ) )
  or
  ( index( ?Res ) = "0.1" and propj01( ?Res ) and propi01( ?Res ) )
  or
  ...
  ( index( ?Res ) = "1.0" and propj10( ?Res ) and propi10( ?Res ) )
  or
  ( index( ?Res ) = "1.1" and propj11( ?Res ) and propi11( ?Res ) )
  or
  ...
  ( index( ?Res ) = "K.0" and propjK0( ?Res ) and propiK0( ?Res ) )
  or
  ( index( ?Res ) = "K.1" and propjK1( ?Res ) and propiK1( ?Res ) )
  or
  ...
)
```

Hence, the options for resources `?res_i` and `?res_j` are joint now. Of course the problem is whether any of such options can be satisfied. However, it is not a problem of the agent; it is a problem of the next service in the workflow. The understanding is always on the side of service or user.

Now, suppose that some service called service2 agreed to realize this intention, and during the workflow execution phase it sent the `message` of `protocolOrder` set as 321 and `protocolSession` set as agent0's address to agent0. The `info` of the `message` is created by service2, so that the address of service2 is put into element `place` of the `info`. The formula in the element `formula` of the `info` is of the following form:

```
(
  ?Res = uri
  and
  token( ?Res ) = Tok
)
```

where `uri` is the `constant` of the same type as the type of `?Res` pointing to the resource produced by service2. Once agent0 has received the `message` of `protocolOrder` set as 321 from service2, it acts as a broker. The agent sends a messages of `protocolOrder` set as 321 to the services in the workflow that need the resource produced by service2. So that agent0 sends the message of `protocolOrder` set as 321 with the formula

```
(
  ?res_j = uri
  and
  token( ?res_j ) = tok0j
)
```

to service1. And at the same time the agent sends a message of `protocolOrder` set as 321 with the formula

```
(
  ?res_i = uri
  and
  token( ?res_i ) = tok0i
)
```

to service1. Service1 downloads the resource denoted by
`?res_j` and sends as the confirmation a message of `protocolOrder` set as 331 and `protocolSession` set as agent0's address to agent0. The `info` of the `message` is created by service1, so that the address of service1 is put into element `place` of the `info`. The formula in the element `formula` of the `info` is of the following form:

```
(
  isIn( ?res_j, service1 )
  and
  token( ?res_j ) = tok0j
)
```

The same is done by service1 for the resource `?res_i` .

Once agent0 has got all the messages of `protocolOrder` set as 331, it sends the confirmation message of `protocolOrder` set as 331 to service2. Once service2 got this message, it replies with a message of `protocolOrder` set as 333 to agent0 that the appropriate intention was realized.

Important note: two or more identical sub terms may occur in the task term not only at the same level of function composition. However, the same procedure is applied also in this case, i.e., these identical sub terms are represented by one term. (This is the job of the agent to analyze the task term and to separate occurrences of the same sub term.) Then, in the course of workflow formation, the agent creates one common intention from the intentions corresponding to the same sub terms. During the workflow execution, the agent acts as a broker and distributes copies of the resource among the services that need it in the workflow.

6.10 Transactional Semantics

If a service cannot realize its `commitment` (i.e., cannot perform its operation and produce the resource it has declared in the commitment), it is obliged to send a `message` of `protocolOrder` set as 009 and `protocolSession` set as agent0's address to agent0. The `info` of the `message` is created by the service, and the formula `( false )` is put into the element `formula` of the `info`.

If agent0 receives a message of `protocolOrder` set as 009 and `protocolSession` set as agent0's address, it is obliged to cancel the distributed transaction associated with the workflow execution. Then, agent0 sends a `message` of `protocolOrder` set as 090 and `protocolSession` set as agent0's address to all the services engaged in this transaction. The element `info` of the `message` is created by agent0. The formula `( false )` is put into the element `formula` of the `info` of the `message`. Once a service has received such message, it performs so called rollback (undo) of the operation it has already performed during the workflow execution. If the service has not performd the operation, then the associated `commitment` formula is removed from the `listOfCommitments` of its `state`.

Once agent0 has canceled the transaction, it removes all formulas from the element `workflow` of its `state`, and starts its routine again from the very beginning, i.e., from the task formula (3.3) if the timeout in the task formula is not over.

Now, suppose that the workflow was performed successfully, i.e., the agent received confirmation messages (of `protocolOrder` set as 333) from all the services engaged in the workflow. Then, agent0 may either send `message` of `protocolOrder` set as 090 and `protocolSession` set as agent0's address that cancels the transaction, or sent synchronously to all the services engaged in the workflow the `message` of `protocolOrder` set as 999 and `protocolSession` set as agent0's address that confirms the distributed transaction. Suppose that the `message` of `protocolOrder` set as 999 and `protocolSession` set as agent0's address was sent. The `info` is created by agent0. The formula `( true )` is put into the element `formula` of the `info`. Once a service has received the `message` of `protocolOrder` set as 999 the effect of performing operation by the service becomes persistent.

If a service has received the `message` of `protocolOrder` set as 090 and `protocolSession` set as agent0's address, the service performs the rollback of the operation it has done during the workflow execution, i.e., the service must undo the temporary results of this operation.

Note, that the main timeout is set in the task formula. This timeout is encoded in the agent's address (see Section 6.1) that is put into the element `protocolSession` of all the messages exchanged in the workflow formation phase and the workflow execution phase. Hence, all services engaged in the workflow know the main timeout. The next timeouts are determined by services in their commitment formulas, and are consecutively earlier and earlier in order to synchronize the workflow execution. Once the main timeout is over and neither the `message` of `protocolOrder` set as 999 and `protocolSession` set as agent0's address nor the `message` of `protocolOrder` set as 090 has been received by the service, the service performs the rollback of the operation it has done during the workflow execution. The transactional semantics implemented in this protocol is similar to the 3PC transaction. It may be seen as is a bit ad hoc solution, however, the transactional semantics is not the main subject of this work. The work concerning the transactions within the framework of enTish technology will be published shortly as dissertation written by Leszek Rozwadowski.

This completes the specification of the protocol entish 1.0.

Chapter 7

Abstract Architecture and Prototype Implementation

In this Chapter an abstract implementation architecture of the enTish technology is presented as well as some details of the prototype implementations.

The first pilot implementation of enTish was completed a year ago, in November 2002. During this implementation and testing, the final versions of the language and the protocol entish 1.0 were refined. Now, the pilot implementation serves as the living demo available on our website.

The prototype implementation of the final version of the language Entish and the protocol entish 1.0 was done by Dariusz Pawluczuk in the framework of his Master thesis under the supervision of the author.

7.1 enTish abstract architecture

There are three basic components of the enTish technology namely, agents, services, and infoServices. The protocol entish 1.0 is a conversation protocol between them. From the point of view of the protocol, a service is a process that represents a raw application in the conversation. An agent is a process that represents a client application, and is dedicated to realization of the task delegated by the client. An infoService represents a service registry.

Although, the agent processes as well as service processes may be implemented individually, i.e., a client may implement its own agent, whereas a service provider may implement its own service process, it is reasonable to have agent server and service server. Agent server is an application dedicated to host agents whereas service server to host service processes. This solution is optimal for several reasons. The first reason is that client applications and service applications may be running inside intranets protected by firewalls, so that they are not visible from outside and cannot participate in a conversations. The second reason is that even if a client application or service applications are down for a while, the corresponding agent or service process can still represent the agent or service in the conversation protocol. Hence, agent server can act as agent proxy whereas service server as service proxy.

There must be special communication protocols between client application and its agent running on an agent server, as well as between service application and the corresponding service process. These protocols are not standardized because there is no need for doing so. They are implementation dependent, i.e., they correspond to a particular abstract architecture of the enTish technology. Anyone who implements a service server and agent server should provide such protocols for client applications and service applications to integrate them with the appropriate server.

Now, we are going to describe the architecture of an application that can be exposed as a service in enTish technology.

7.1.1 Service architecture

We follow the idea of a layered view of service architecture introduced in (15; 63). Our service architecture comprises the following three layers: Conversation layer, functionality layer, and database management (executive) layer. The database management layer is the same as in (15), it influences the real world. However, the next two layers have different meaning. The functionality layer has exactly two interrelated components: Raw application, and so called filter associated with the raw application. Raw application implements a single operation, i.e., given input resources, it produces the output resource according to the operation specification. Note, that operation has exactly one output, although it may have several inputs. The associated filter works as follows. Given constrains on the output resource, it produces the constrains on the input resources. That is, given a specification of the desired output, the filter replies with properties that must be satisfied by the input in order to produce the desired output by the raw application. It is clear that these constrains must be expressed in the description language, i.e., in Entish. The conversation layer implements the communication protocol between the application and a service server in order to arrange raw application invocation, to pass input / output resource to / from the raw application, as well as to realize transactional semantics.

Since our service architecture is different than the one corresponding to WSDL and UDDI of the Web services, we were forced to revise the concept of service description language as well as the concept of service registry. It is natural that service description language should describe the types of service input / output resources as well as attributes of these resources to express constrains. Note, that the description language is supposed merely to *describe* resources in terms of theirs attributes, not to construct data structures as it is done in WSDL. It is also natural to describe *what service does* in the language, i.e., to describe the type of the operation the service performs. This type is expressed in terms of abstract function implemented by the operation. In Web services technology, *what service does* is described in UDDI. We express this description in our language Entish.

Since service has additional functionality performed by filter (i.e., service may be asked if it can produce output resources satisfying some properties), the description language Entish was augmented with a possibility to formulate such questions as well as answers.

Note that Entish describes also some static properties of service composition process such as intentions, and commitments; this corresponds to the functionality of WS-Coordination (41).

7.1.2 The demo of the pilot implementation

Let us present the demo of the pilot implementation. The system resulting from this implementation allows providers to join their application as services as well as to formulate requests by clients, and delegate the requests to the system for realization.

The demo of the system is available for testing and evaluation via three www interfaces starting with http://www.ipipan.waw.pl/mas/ . The first interface called *EntishDictionary* serves for introducing names for new data types, their attributes, and new functions to the language Entish. The second interface called *serviceAPI* is for joining applications (that implement the new functions) as networked services to our system. The third interface called TaskManager is devoted to a client to specify its request (task) in Entish, and provide initial resources for the task realization. Hence, from the outside, i.e., from service providers and clients point of view, the system consists of the three interfaces: EntishDictionary, TaskManager, and serviceAPI. What is inside, that is, the system engine is transparent for service providers and clients. The engine implements the language Entish, and the protocol entish 1.0 that realizes clients' tasks by discovering appropriate services, composing them into a workflow, and finally invoking them. The www interfaces are user friendly so that to use the system almost no knowledge on XML, the Entish syntax, and the protocol entish 1.0 is needed.

EntishDictionary (ED for short) serves also as an ordinary dictionary, i.e., for looking at the existing ontologies as well as for explanation of names used in the language. Ontology is meant as a collection of names of resource types, their attributes, and functions defined on these types. It is supposed that they come from one application domain. Any user can introduce its own ontology. There is no conflict of names, because short names introduced by users are automatically extended to long names that are URIs (64). ED has also additional functionality. It allows a service provider to create Entish formula that describes the type of operation performed by its application. The formula along with some additional information about the host on which the application is running is sent automatically to the system (actually to serviceServer) for registration and publication.

The second functionality of ED is that it allows the user to create a task; usually it is a composition of abstract functions. The task is sent automatically to TaskManager for realization. It is important to note, that a task is merely an abstract description of what is to be realized, so that it does not indicate what services could realize it. Note, that EntishDictionary realizes some functionality of UDDI concerning service classification. However, service publication and discovery are done inside the system.

Generally in enTish, there is no restriction on the type of resources that can be defined in the dictionary. However, for the purpose of system demonstration we assume that a resource type has flat XML format, i.e., it has several elements that are of type `xsd:string`.

TaskManager is a GUI that, given a task from a user, creates and manages appropriate interfaces for delivering (by the user) initial resources needed for task realization. Then, the task is sent to the system (actually to *agentServer*) for realization. If the system is ready to realize the task (it means that a workflow consisting of appropriate services has already been arranged), the TaskManager asks the user to provide the initial resources according to the constrains returned by the system. More sophisticated tasks can be generated directly from a repository of typical tasks provided by the TaskManager.

The interface serviceAPI provides Java classes and explanation for creating services (according to our architecture) by a service provider.

7.1.3 The prototype

The prototype implementation is based on the same abstract architecture and realizes the same functionality as the demo presented above, however without such graphical interfaces. In our opinion, it verifies the enTish technology in the sense that the description language and the composition protocol can be applied to integration of heterogeneous applications in open and distributed environment based on HTTP as the transport protocol between the applications.

The fact that the enTish technology can be realized within the framework of a Master thesis (notabene a very good thesis) proves that enTish is not too complex. However, more independent implementations are needed to verify the interoperablity between components.

7.2 Conclusion and summary of the work

enTish is not based on the basic protocol stack of the Web services, i.e., on SOAP+WSDL+UDDI. It may be seen as the main drawback of enTish. All existing technologies that aim at integrating heterogeneous services in open environment are based on the Web services or provide translation to the Web services like DAML-S.

Actually enTish is based on different principles, so that it is impossible to translate it into Web services. The main difference is the service description language. WSDL is an interface definition language, and provides means for binding service interface with a concrete application, and with a transport protocol.

Entish describes service operation types in terms of abstract functions implemented by operations, and in terms of precondition and post condition of operation invocation. Similar approach was taken by DAML-S, however, it is based on RDF, and therefore it is enormously complex.

As to SOAP it has turned out recently (in SOAP 1.2) that only asynchronous document passing style of communication between services is obligatory. Data that are passed between applications are included in documents. Since RPC-style is optional, it means that it is not considered as a communication method any more.

In enTish the asynchronous message passing as well as separate asynchronous data passing are used. Hence, the enTish communication cannot be reduced to SOAP.

UDDI as a service registry has a lot of functionality useful for business applications. However, from the point of view of integrating heterogeneous services it is not sufficient. It seems that the main reason is that the taxonomy (provided by UDDI) for classifying service operations is not generic and open.

In enTish, a service registry is realized as infoService. It is extremely simple and useful for our experimental technology.

Since enTish is based on different principles, it should be viewed as an alternative approach to integrating heterogeneous services.

Since enTish is an experimental technology, it is not perfect. In fact, the main reason for designing and realizing enTish was to prove that it is possible to integrate heterogeneous applications in open and distributed environment, and provide a simple solution for realizing it.

One of the key aspects that are only mentioned in the work is transactional semantics implemented in the protocol entish 1.0. Obviously, there are other possible ways for doing so. It would be interesting to compare the transactional semantics implemented in enTish with the ones proposed for example by BPEL4WS, WS-Coordination, WS-Transactions. This, however, is the subject of dissertation prepared by Leszek Rozwadowski - a member of the enTish team.

Another important aspect is the concept of Entish Dictionary as an open and distributed ontology. Also in this case a detailed discussion and comparison with existing approaches to ontologies, especially to DAML and OWL would be extremely interesting. However, this is the subject of dissertation prepared by Dariusz Mikulowski - another member of the enTish team.

Bibliography

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer-Verlag 2004.

[2] Ambroszkiewicz S. enTish: An Approach to Service Composition. In B. Benatallah and M-Ch. Shan (Eds.), Technologies for E-Services. Proc. of 4th Int. Workshop TES 2003, Berlin, Sept. 2003, by Springer LNCS 2819, (2003), ISBN 3-540-20052-5, 168-178.

[3] Ambroszkiewicz S. Entish: a simple language for Web Service Description and Composition, In (eds.) W. Cellary and A. Iyengar. Internet Technologies, Applications and Societal Impact. Kluwer Academic Publishers, (2002), 289-306.

[4] Ambroszkiewicz S., Nowak T., Mikulowski D., and Rozwadowski L. A Concept of Agent Language in Agentspace. In From Theory to Practice in Multi-Agent Systems. Springer LNAI 2296. pp. 37-46. ISBN: 3-540-43370-8.

[5] Ambroszkiewicz S. Web Service Integration as a New Paradigm for Networked Computing. In Proc. PARELEC-2002, International Conference on Parallel Computiong and Electrical Engineering. 22-25 September 2002, Warsaw, Poland. IEEE Computer Society Press, ISBN 0-7695-1730-7, pp. 239-245.

[6] Ambroszkiewicz S. Semantic Interoperability in Agentspace: an approach to concept meaning. In Kangassalo H., Jaakkola H., (Eds.) Information Modelling and Knowledge Bases XII, IOS Press, Amsterdam, pp. 151-161, (2001). ISBN 1-58603-163-5.

[7] Ambroszkiewicz S., and Nowak T. Agentspace as a Middleware for Service Integration. In Proc. ESAW'2001. Springer-Verlag LNAI, vol. 2203, ISBN 3-540-43092-1, pp. 134-159.

[8] Austin, J. L. *How to Do Things with Words.* Cambridge, Mass.: Harvard University Press, (1962).

[9] B. Benatallah, M. Dumas, and Q. Z. Sheng. The Self-Serv Environment for Web Services Composition. IEEE Internet Computing, Jan. - Feb. 2003, pp. 40 - 48.

[10] T. Berners-Lee, James Hendler, Ora Lassila, The Semantic Web, Scientific American, May 2001

[11] D. Gelernter. Generative communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1):-112, January 1985.

[12] E. Husserl. Die Krisis der europischen Wissenschaften und die transzen-

tale Phnomenologie: Eine Einleitung in die phnomenologische Philosophie," Philosophia 1 (1936) 77-176.

[13] I. Kant. *Critique of Pure Reason*, trans. Werner Pluhar. Indianapolis: Hackett, 1996

[14] F. Kaufmann. *Das Unendliche in der Mathematik und seine Ausschaltung.* Leipzig / Wien, 1930

[15] F. Leymann and D. Roller. Workflow-based applications. IBM Systems Journal, Volume 36, Number 1, 1997 Application Development http://researchweb.watson.ibm.com/journal/sj/361/leymann.html

[16] Nelson B. J. Birrell A. D. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1), February 1984.

[17] A. Oram (editor) *Peer to Peer: Harnessing the Power of Disruptive Technologies,* by O'Reilly & Associates, Inc. First Edition March (2001).

[18] Ch. S. Peirce. On a New List of Categories. Proceedings of the American Academy of Arts and Sciences 7 (1868), 287-298.

[19] A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI–architecture. In R. Fikes and E. Sandewall, editors, *Proc. of the 2rd International Conference on Principles of Knowledge Representation and Reasoning (KR'91),* Morgan Kaufmann, (1991), pp. 473-484.

[20] Searle, J. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge, Eng.: Cambridge University Press, (1969).

[21] Szyperski, C. *Component-Oriented Programming A Refined Variation on Object-Oriented Programming.* The Oberon Tribune, Vol 1, No 2, December 1995, http://www.oberon.ch/resources/component_software/cop.html

[22] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley (1998).

[23] Tarski A., *Pojęcie prawdy w językach nauk dedukcyjnych*, Prace Towarzystwa Naukowego Warszawskiego (Wydzial III nauk matematyczno-fizycznych), 1933

[24] J.R.R. Tolkien. *The Lord of the Rings.* III ch. 4

[25] L. Wittgenstein, *Philosophical Investigations.* Basil Blackwell, Oxford 1958, pp. 20-21.

[26] KQML - Knowledge Query and Manipulation Language, http://www.cs.umbc.edu/kqml/

[27] FIPA ACL http://www.fipa.org

[28] OSF DCE 1.0 Application Development Guide. Technical report, Open Software Foundation, December 1991.

[29] ISO91 ISO Remote Procedure Call Specification. ISO/IEC CD 11578 N6561, ISO/IEC, November 1991.

[30] SUN90 Sun Microsystems Inc. Network Programming Guide, Revision A March 27 1990.

[31] UN91 Solaris ONC: Design and Implementation of Transport-Independent RPC. Solaris 2.0 White Papers, SunSoft, 1991.

[32] OMG CORBA, Common Object Request Broker Architecture. http://www.corba.org/

[33] The Linda group homepage. Online. Department of Computer Science, Yale University. Available: http://www.cs.yale.edu/HC1L/YALE/CS/Linda/linda.html.

[34] Jini/Javaspaces language by Sun Microsystems http://wwws.sun.com/software/jini/

[35] IBM TSpaces http://www.almaden.ibm.com/cs/TSpaces/

[36] D. Tidwell. Web services: the Web's next revolution. Nov. 2000, http://www-106.ibm.com/developerworks/edu/ws-dw-wsbasics-i.html

[37] SOAP Version 1.2, W3C Recommendation 24 June 2003, http://www.w3.org/TR/soap12-part1/

[38] Web Services Description Language (WSDL) Version 1.2, W3C Working Draft 11 June 2003, http://www.w3.org/TR/wsdl12/

[39] Universal Description, Discovery and Integration (UDDI), http://www.uddi.org/

[40] Business Process Execution Language for Web Services Version 1.1 http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[41] Web Services Coordination (WS-Coordination), http://www-106.ibm.com/developerworks/library/ws-coor/

[42] Web Services Transaction (WS-Transaction), http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/

[43] Web Services Business Process Execution Language TC of OASIS, http://www.oasis-open.org/committees/wsbpel/

[44] DAML-S is DAML-based Web Service Ontology, http://www.daml.org/services/

[45] Web Service Choreography Interface (WSCI) 1.0 www.w3.org/TR/wsci/

[46] Web Services Choreography Working Group of W3C http://www.w3.org/2002/ws/chor/

[47] The Buisiness Process Management Language (BPML), www.bpmi.org

[48] Web Services Composite Application Framework (WS-CAF) http://developers.sun.com/techtopics/webservices/wscaf/

[49] M. Papazoglou, M. Aiello, M. Pistore, and J. Yang. XSRL: A Request Language for Web Services. http://www.webservices.org/index.php/article/articleview/990/1/24/

[50] GGF - The Global Grid Forum, http://www.ggf.org/

[51] J. Unger and M. Haynos. A visual tour of Open Grid Services Architecture. IBM developerWorks, http://www-106.ibm.com/developerworks/grid/library/gr-visual/

[52] Open Grid Services Architecture (OGSA) http://www-fp.globus.org/ogsa/

[53] S. Weerawarana. WSDL 1.2: Proposed resolution to portType extensibility and service type issues. http://lists.w3.org/Archives/Public/www-ws-desc/2002Jun/att-0046/01-portTypes-2002-06-09.html

[54] Web Services Message Exchange Patterns, W3C draft http://www.w3.org/2002/ws/cg/2/07/meps.html

[55] W3C Semantic Web, http://www.w3.org/2001/sw/

[56] Resource Description Framework, http://www.w3.org/RDF/

[57] S. Hawke. How the Semantic Web Works. http://www.w3.org/2002/03/semweb/

[58] A. Swartz. The Semantic Web In Breadth. http://logicerror.com/semanticWeb-long

[59] S. B. Palmer. The Semantic Web: An Introduction. http://infomesh.net/2001/swintro/

[60] RDF Vocabulary Description Language 1.0: RDF Schema, http://www.w3.org/TR/rdf-schema/

[61] DARPA Agent Markup Language http://www.daml.org

[62] OWL Web Ontology Language http://www.w3.org/TR/owl-features/

[63] S. Kumaran and P. Nandi. Conversational Support for Web Services: The next stage of Web services abstraction.
http://www-106.ibm.com/developerworks/webservices/library/ws-conver/

[64] Naming and Addressing: URIs, URLs, ... http://www.w3.org/Addressing/

Appendix A

XML Sources of the enTish Technology

The XML version of the Entish syntax presented in this chapter was created in cooperation with Dariusz Mikulowski.

The three schemes `message.xsd, state.xsd`, and `info.xsd` define the communication Entish, i.e., the data structures for `message`, `state`, and `signedInfo`. Messages are exchanged between conversation parties, (i.e., agents and services) according to a fixed protocol. Message exchange may change the state of the sender and the recipient. The basic element of `message` and `state` is `signedInfo` defined in `info.xsd`. The meaning of an element `signedInfo` is that the Entish formula (the basic component of `signedInfo`) was true at some time and in a place.

Two documents `formula.xsd` and `definitions.xsd` constitute the upper Entish that is nothing but an empty form for expressing well formed formulas.

The upper Entish is a simple version of the language of first order logic with types, and without quantifiers. The XML version of the upper Entish syntax is defined in `formula.xsd` schema, where the definition of well formed formulae is introduced. The language is open. It means that names of new primitive concepts, i.e., names for new types of resources, names for their attributes, names for new relations, as well as names for new functions can be introduced to the language by any user. This can be done by an instance of the schema `definitions.xsd`. The name of any new concept must be uniqe and contain the URI to the very document where it was defined.

The document `properEntish.xml` defines the standard part of the language Entish. It is introduced in the same way as another concepts (ontologies) can be introduced, i.e., `properEntish.xml` is an instance of the schema `definitions.xsd`.

The properEntish may be considered as a proposal of upper ontology for service description and composition.

A.1 XML-schema message.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://ii5.ap.siedlce.pl/entish"
      xmlns="http://ii5.ap.siedlce.pl/entish"
      elementFormDefault="qualified">
    <xsd:include schemaLocation="info.xsd"/>
<!--
    Xsd schema defining message format in the protocol entish.
    Authors: S. Ambroszkiewicz, D. Mikulowski, and D. Pawluczuk
    http://www.ipipan.waw.pl/mas
    Last modified April 2, 2003
-->
  <xsd:element name="message">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="header">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="from" type="xsd:anyURI"/>
                <xsd:element name="to" type="xsd:anyURI"/>
                <xsd:element name="protocolName" type="xsd:string"/>
                <xsd:element ref="protocolVersion"/>
                <xsd:element name="protocolSession" type="xsd:string"/>
                <xsd:element ref="protocolOrder"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="body" type="listOfInfos"/>
       </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="protocolVersion">
    <xsd:simpleType>
       <xsd:restriction base="xsd:string">
         <xsd:pattern value="\d{1}.\d{2}"/>
       </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>

  <xsd:element name="protocolOrder">
    <xsd:simpleType>
      <xsd:restriction base="xsd:nonNegativeInteger">
        <xsd:maxInclusive value="999"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>

</xsd:schema>
```

A.2 XML-schema info.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://ii5.ap.siedlce.pl/entish"
      xmlns="http://ii5.ap.siedlce.pl/entish"
      elementFormDefault="qualified">
    <xsd:include schemaLocation="formula.xsd"/>
<!--
    Xsd schema defining fact, i.e., Entish evaluated formula.
    Authors: S. Ambroszkiewicz, D. Mikulowski, and D. Pawluczuk
    http://www.ipipan.waw.pl/mas
    Last modified April 2, 2003
-->
 <xsd:element name="signedInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="info"/>
        <xsd:element name="signature" type="xsd:anyURI" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="info">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="formula"/>
        <xsd:element name="place" type="xsd:anyURI"/>
        <xsd:element name="time" type="xsd:dateTime"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="listOfInfos">
    <xsd:sequence>
      <xsd:element ref="signedInfo" maxOccurs="unbounded" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

A.3 XML-schema state.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ii5.ap.siedlce.pl/entish"
    xmlns="http://ii5.ap.siedlce.pl/entish"
    elementFormDefault="qualified">
    <xsd:include schemaLocation="info.xsd"/>
    <xsd:include schemaLocation="formula.xsd"/>
 <!--
    Xsd schema defining agent’s / service’s state in the protocol entish.
    Authors: S. Ambroszkiewicz, D. Mikulowski, and D. Pawluczuk
    http://www.ipipan.waw.pl/mas
    Last modified April 2, 2003
-->
  <xsd:element name="state">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="owner" type="xsd:anyURI"/>
        <xsd:element ref="goal"/>
        <xsd:element ref="intentions"/>
        <xsd:element ref="listOfCommitments"/>
        <xsd:element name="knowledge" type="listOfInfos"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="goal">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="formIn" type="Formula" minOccurs="0"/>
        <xsd:element name="formOut" type="Formula"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="intentions">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="plan" type="listOfFormulas"/>
        <xsd:element name="workflow" type="listOfFormulas"/>
        <xsd:element name="realized" type="listOfFormulas"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="listOfFormulas">
    <xsd:sequence>
      <xsd:element ref="formula" maxOccurs="unbounded" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="listOfCommitments">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="commitment" maxOccurs="unbounded" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="commitment">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="formIn" type="Formula"/>
        <xsd:element name="formOut" type="Formula"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

A.4 XML-schema formula.xsd

```
<?xml version="1.0"?>
<!--
    Xsd schema defining Entish formula and term.
    Authors: S. Ambroszkiewicz, D. Mikulowski, D. Pawluczuk,
    M. Calka, and P. Izdebski
    http://www.ipipan.waw.pl/mas
    Last modified April 2, 2003
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://ii5.ap.siedlce.pl/entish"
 xmlns="http://ii5.ap.siedlce.pl/entish"
 elementFormDefault="qualified">

  <xsd:element name="formula" type="Formula"/>
  <xsd:complexType name="Formula">
    <xsd:sequence>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element ref="relationName"/>
          <xsd:element ref="term" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="operator">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:enumeration value="or"/>
                <xsd:enumeration value="and"/>
                <xsd:enumeration value="implies"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element ref="formula" minOccurs="2" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
<!--
    Note, the following three elements of type Concept MUST correspond to
    the appropriate elements of relationDefinition, functionDefinition and
    typeDefinition defined in definitions.xsd. Moreover, the signature of
    relationName, or functionName MUST agree with the signature defined in
    the relationDefinition, or functionDefinition. The element longName of
    relationName, or functionName, or typeName MUST be the URI of
    the document (an instance of definitions.xsd) where the Name
    was defined.
-->
  <xsd:element name="relationName" type="Concept"/>
  <xsd:element name="functionName" type="Concept"/>
  <xsd:element name="typeName" type="Concept"/>
  <xsd:complexType name="Concept">
    <xsd:sequence>
```

```
      <xsd:element name="shortName" type="xsd:string"/>
      <xsd:element name="longName" type="xsd:anyURI" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="term" type="Term"/>
  <xsd:complexType name="Term">
    <xsd:sequence>
      <xsd:choice>
        <xsd:sequence>
          <xsd:choice>
            <xsd:element ref="variable"/>
            <xsd:element ref="constant"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element ref="functionName"/>
          <xsd:element ref="typeName"/>
          <xsd:element ref="term" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="variable" type="Variable"/>
  <xsd:complexType name="Variable">
    <xsd:sequence>
      <xsd:element name="variableName" type="xsd:string"/>
      <xsd:element ref="typeName" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="constant" type="Constant"/>
  <xsd:complexType name="Constant">
    <xsd:sequence>
      <xsd:element name="constantName" type="Concept"/>
      <xsd:element ref="typeName"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

A.5 XML-schema definitions.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://ii5.ap.siedlce.pl/entish"
 xmlns="http://ii5.ap.siedlce.pl/entish"
 elementFormDefault="qualified">
 <xsd:include schemaLocation="formula.xsd"/>
<!--
    The schema is for introducing new concepts, i.e., types, relation,
    and function names to the language Entish.
    Authors: S. Ambroszkiewicz, D. Mikulowski, and D. Pawluczuk
    http://www.ipipan.waw.pl/mas
    Last modified April 2, 2003
-->
 <xsd:element name="definitions">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="typeDefinition" minOccurs="0"
                                            maxOccurs="unbounded"/>
      <xsd:element ref="relationDefinition" minOccurs="0"
                                            maxOccurs="unbounded"/>
      <xsd:element ref="functionDefinition" minOccurs="0"
                                            maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="typeDefinition">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="definiens">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="typeName"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="definiendum">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="documentation"/>
           </xsd:sequence>
     </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

<!--
    Note that in relationDefinition as well as in
    functionDefinition specified below the variables
    (if there are any) occurring in the element formula
    (resp. element term) of the element definiendum,
    MUST be the same as all variables occurring in the
    element definiens.
-->

 <xsd:element name="relationDefinition">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="definiens">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="relationName"/>
             <xsd:element ref="variable" minOccurs="0"
                                         maxOccurs="unbounded"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="definiendum">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="formula" minOccurs="0"/>
             <xsd:element ref="documentation"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

 <xsd:element name="functionDefinition">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="definiens">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="functionName"/>
             <xsd:element ref="typeName"/>
             <xsd:element ref="variable" minOccurs="0"
                                         maxOccurs="unbounded"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="definiendum">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element ref="term" minOccurs="0"/>
             <xsd:element ref="documentation"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>
 <xsd:element name="documentation">
   <xsd:complexType>
     <xsd:sequence>
       <xsd:element name="humanInfo" type="xsd:string"/>
       <xsd:element name="appiInfo" minOccurs="0">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:any/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

</xsd:schema>
```

A.6 XML-schema properEntish.xml

```
<?xml version="1.0"?>
<definitions xmlns="http://ii5.ap.siedlce.pl/entish"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ii5.ap.siedlce.pl/entish
  definitions.xsd">
<!--
     Introduction of basic Entish primitive types, relations, and functions.
     Last modified April 9, 2003.
     Authors: S. Ambroszkiewicz and D. Mikulowski.
-->

  <typeDefinition>
    <definiens>
      <typeName>
        <shortName>Agent</shortName>
        <longName>
          http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#Agent
        </longName>
      </typeName>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          agent (i.e., element of type Agent) is a process equipped with
          state (i.e., element State defined in state.xsd).
          A constant (defined in formula.xsd) of this type contains
          agent's short name in the element shortName,
          and agent's communication address in the element longName.
          It is supposed that all essential data of agent are stored in
          its state.  Agent is dedicated for a single task realization.
          It is created when there is a task to be realized, and is
          terminated after the task realization or if the task can not
          be realized. More details on http://www.ipipan.waw.pl/mas/
        </humanInfo>
      </documentation>
    </definiendum>
  </typeDefinition>

  <typeDefinition>
    <definiens>
      <typeName>
        <shortName>Service</shortName>
        <longName>
         http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#Service
        </longName>
      </typeName>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          service (i.e., element of type Service) is a process having
        its own state (i.e., element State defined in state.xsd).
        It is an application processing data (e-documents).
        A constant (defined in formula.xsd) of this type contains
        service's short name in the element shortName, and service's
        communication address in the element longName.
        Processing e-documents may result in effecting the real world,
        e.g., purchasing a commodity or withdraw of some amount of
        money from a bank account, or just taking some physical actions
        like switching off/on a washing machine.
        More details on http://www.ipipan.waw.pl/mas/
      </humanInfo>
    </documentation>
  </definiendum>
</typeDefinition>

<typeDefinition>
  <definiens>
    <typeName>
      <shortName>Time</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#Time
      </longName>
    </typeName>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
        element of this type is a date and time written according to
        xsd:dateTime format. A constant (defined in formula.xsd) of
        this type contains this string in the element shortName of
        the constantName, and there is no longName in the constantName.
      </humanInfo>
    </documentation>
  </definiendum>
</typeDefinition>

<typeDefinition>
  <definiens>
    <typeName>
      <shortName>Token</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#Token
      </longName>
    </typeName>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
        token is an arbitrary string of type xsd:string.
        A constant (defined in formula.xsd) of this type contains
        this string in the element shortName of the constantName,
        and there is no longName in the constantName. It is used as
        value of function token(?resource). It is a general way to
        identify resources (e-docs) of any type on the language level.
        Note that our language Entish is independent from data format
        of resources; the format may be arbitrary, e.g., MS Word, XML,
        txt, binary, and so on.
      </humanInfo>
    </documentation>
  </definiendum>
</typeDefinition>

<typeDefinition>
  <definiens>
    <typeName>
      <shortName>Index</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#Index
      </longName>
    </typeName>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
        element of type Index is a string of type xsd:string.
        The string is the form of decimal numbers
        separetad by dots in the same way as IP addresses, e.g.,
        '0.12.3'
        '34.0.11.45.1'
        '5.34'
        A constant (defined in formula.xsd) of this type contains
        this string in the element shortName of the constantName, and
        there is no longName in the constantName. It is used as value
        of function index( ?resource ). The value (index) is
        associated with an option. Indexes serve to determine
        interdependencies between options that describe resources
        in workflow.
      </humanInfo>
    </documentation>
  </definiendum>
</typeDefinition>

<relationDefinition>
  <definiens>
    <relationName>
      <shortName>intentions</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#intentions
      </longName>
    </relationName>
    <variable>
      <variableName>?agent</variableName>
      <typeName>
        <shortName>Agent</shortName>
      </typeName>
    </variable>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          intentions(?agent) is an atomic formula.
          It is evaluated (only!) locally (i.e., by ?agent).
          The evaluation returns the disjunction of all formulas
          from the element plan of the state of the ?agent.
        </humanInfo>
      </documentation>
    </definiendum>
  </relationDefinition>

  <relationDefinition>
    <definiens>
      <relationName>
        <shortName>true</shortName>
        <longName>
          http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#true
        </longName>
      </relationName>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          true is the relation always true,
          it corresponds to the Boolean value "true".
        </humanInfo>
      </documentation>
    </definiendum>
  </relationDefinition>

  <relationDefinition>
    <definiens>
      <relationName>
        <shortName>false</shortName>
        <longName>
          http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#false
        </longName>
      </relationName>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          false is the relation always false,
          it corresponds to the Boolean value "false".
        </humanInfo>
      </documentation>
    </definiendum>
  </relationDefinition>

  <relationDefinition>
    <definiens>
```

```
      <relationName>
        <shortName>timeout</shortName>
        <longName>
          http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#timeout
        </longName>
      </relationName>
      <variable>
        <variableName>
         ?t
        </variableName>
        <typeName>
          <shortName>Time</shortName>
        </typeName>
      </variable>
 </definiens>
 <definiendum>
    <documentation>
      <humanInfo>
        timeout(?t) can be evaluated at any host.
        It is true if the time ?t is less or equal to the
        current GMT time at the host.
      </humanInfo>
    </documentation>
  </definiendum>
</relationDefinition>

<relationDefinition>
  <definiens>
    <relationName>
      <shortName>equals</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#equals
      </longName>
    </relationName>
    <variable>
      <variableName>?x</variableName>
    </variable>
    <variable>
      <variableName>?y</variableName>
    </variable>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
        equals(?x, ?y) or more frequently (?x = ?y) is a polimorphic
        equality relation.
        It can be evaluated if ?x and ?y are of the same type.
      </humanInfo>
    </documentation>
  </definiendum>
</relationDefinition>

<relationDefinition>
```

```
    <definiens>
        <relationName>
          <shortName>formInOperationType</shortName>
          <longName>
            http://ii5.ap.siedlce.pl:8080/Entish/
                                properEntish.xml#formInOperationType
          </longName>
        </relationName>
      <variable>
        <variableName>?service</variableName>
        <typeName>
          <shortName>Service</shortName>
        </typeName>
      </variable>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          formInOperationType( ?service ) is an atomic formula to
          be evaluated only by ?service. The evaluation returns the
          formula from the formIn element of the state of ?service.
          The formula describes the precondition necessary for
          ?service invocation.
        </humanInfo>
      </documentation>
    </definiendum>
  </relationDefinition>

  <relationDefinition>
    <definiens>
        <relationName>
          <shortName>formOutOperationType</shortName>
          <longName>
            http://ii5.ap.siedlce.pl:8080/Entish/
                                properEntish.xml#formOutOperationType
          </longName>
        </relationName>
      <variable>
        <variableName>?service</variableName>
        <typeName>
          <shortName>Service</shortName>
        </typeName>
      </variable>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          formOutOperationType( ?service ) is an atomic formula to
          be evaluated by ?service. The evaluation returns the
          formula from the formOut element of the state of ?service.
          The formula describes the postcondition of ?service
          invocation, i.e., the result of performing the operation
          by ?service.
      </humanInfo>
    </documentation>
  </definiendum>
</relationDefinition>

<relationDefinition>
  <definiens>
    <relationName>
      <shortName>formInCommitment</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/
                               properEntish.xml#formInCommitment
      </longName>
    </relationName>
    <variable>
      <variableName>?service</variableName>
      <typeName>
        <shortName>Service</shortName>
      </typeName>
    </variable>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
        formInCommitment( ?service ) is an atomic formula evaluated
        only by ?service. Evaluation returns the disjunction of
        formulas from formIn elements of all commitment elements of
        the state of ?service. It describes the precondition of the
        commitments made by the ?service.
      </humanInfo>
    </documentation>
  </definiendum>
</relationDefinition>

<relationDefinition>
  <definiens>
      <relationName>
        <shortName>formOutCommitment</shortName>
        <longName>
          http://ii5.ap.siedlce.pl:8080/Entish/
                                 properEntish.xml#formOutCommitment
        </longName>
      </relationName>
    <variable>
      <variableName>?service</variableName>
      <typeName>
        <shortName>Service</shortName>
      </typeName>
    </variable>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
```

```
        formOutCommitment( ?service ) is an atomic formula evaluated
        only by ?service. Evaluation returns the conjunction of
        formulas from formOut elements of all commitment elements of
        the state of ?service. It describes the postconditions of the
        commitments made by the ?service
      </humanInfo>
    </documentation>
  </definiendum>
</relationDefinition>

<relationDefinition>
  <definiens>
    <relationName>
      <shortName>isIn</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#isIn
      </longName>
    </relationName>
    <variable>
      <variableName>?resource</variableName>
    </variable>
    <variable>
      <variableName>?place</variableName>
      <typeName>
        <shortName>Service</shortName>
      </typeName>
      <typeName>
        <shortName>Agent</shortName>
      </typeName>
    </variable>
  </definiens>
  <definiendum>
    <documentation>
      <humanInfo>
        isIn( ?resource, ?place )
        states that  ?resource is in ?place.
        It can be evaluated only in ?place.
        Usually, the ?place denotes either service or agent.
      </humanInfo>
    </documentation>
  </definiendum>
</relationDefinition>

<functionDefinition>
  <definiens>
    <functionName>
      <shortName>token</shortName>
      <longName>
        http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#token
      </longName>
    </functionName>
    <typeName>
      <shortName>Token</shortName>
      </typeName>
      <variable>
        <variableName>?resource</variableName>
      </variable>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          function token( ?resource ) returns token determined for
          ?resource.
        </humanInfo>
      </documentation>
    </definiendum>
  </functionDefinition>

  <functionDefinition>
    <definiens>
      <functionName>
        <shortName>index</shortName>
        <longName>
          http://ii5.ap.siedlce.pl:8080/Entish/properEntish.xml#index
        </longName>
      </functionName>
      <typeName>
        <shortName>Index</shortName>
      </typeName>
      <variable>
        <variableName>?resource</variableName>
      </variable>
    </definiens>
    <definiendum>
      <documentation>
        <humanInfo>
          function index( ?resource ) returns an index that is
          associated with an option. Options describe the resources
          in workflow. Indexes serve to determine interdependencies
          between the options in workflow.
        </humanInfo>
      </documentation>
    </definiendum>
  </functionDefinition>

</definitions>
```