INF-3981

MASTER'S THESIS IN COMPUTER SCIENCE

A Prototype for Continuation-Passing Orchestration of Composite Web Services

Kent Ove Josefsen

December 15th, 2007

FACULTY OF SCIENCE
Department of Computer Science

University of Tromsø

INF-3981

MASTER'S THESIS IN

COMPUTER SCIENCE

A Prototype for Continuation-Passing Orchestration of Composite Web Services

Kent Ove Josefsen

December 15th, 2007

Abstract

Web services enable businesses to deliver services via the Web. In addition, Web services can be used to improve business efficiency, because Web services can often replace manual activities in a business process. A Web service can be composed of other simpler Web services using the abstraction traditionally known as workflows. The process of controlling the correct and reliable execution of a workflow is known as workflow enactment, but it can also be referred to as Web services orchestration.

A centralized approach to Web services orchestration is not an optimal solution, when considering the decentralized nature of Web services and the Internet. The problem is that a centralized workflow engine can easily become a communication and processing bottleneck as well as a central point of failure. Therefore, a decentralized approach to Web services orchestration is needed in order to overcome these obstacles. However, some decentralized approaches require static analysis of workflow specifications, which implies that resources must be allocated prior to the workflow execution. As a result, resources are wasted.

In this thesis a prototype is developed to demonstrate a decentralized Web services orchestration based on continuation-passing enactment of distributed workflows. This decentralized approach does not require any pre-allocation of resources, nor is it subject to the limitations of a centralized approach. The continuation-passing mechanism involves continuations, or the reminder of the executions, which are passed along asynchronous messages for workflow enactment.

Acknowledgements

I would like to thank my supervisor Weihai Yu at the University in Tromsø, who helped me find the topic for this thesis and who gave me assistance.

I would also like to thank the Apache community for providing the necessary tools which made it possible to realize the prototype.

Finally, I would like to thank my family and friends for their moral support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As businesses evolve, so does business software. Or, to put it another way, business software helps businesses to evolve. However, business software relies heavily on information technology (IT). This often leads to problems when businesses need to integrate their applications, because the IT systems they use are often heterogeneous. The motivation for integrating business software is often to improve business efficiency.

Java [16] can be used as an enterprise development platform, where an enterprise corresponds to a large business, in order to remove some of the issues with heterogeneous hardware. This is mainly because Java byte code will run anywhere the Java Virtual Machine (JVM) runs. The version of Java suitable for enterprises is called Java 2 Platform, Enterprise Edition (J2EE). However, an alternative to J2EE exists from Microsoft which is called .NET, but .NET applications can not run on a platform from a vendor other than Microsoft. Therefore, it can be said that J2EE portability is good, while .NET portability is limited. However, interoperability is in some cases more important than portability. Interoperability between J2EE applications and .NET applications can be achieved through Enterprise Application Integration (EAI).

One approach to EAI is to use Message-Oriented Middleware (MOM), where the solutions are based on message brokers. The downside of using this approach is that MOM only works best inside of a firewall [21], and it also requires the investment in expensive software licenses [6]. In addition, message brokers requires adapters which adds more to the cost.

Another approach to EAI is to use Web services technology [19]. In this thesis, a *Web service* will correspond to a service that is described via Web Service Description Language (WSDL) and that is capable of being accessed via standard network protocols, such as SOAP over HyperText Transport Protocol (HTTP). Because a Web service is transport neutral, it has actually nothing to do with the Web. The benefit of using HTTP to transport SOAP messages is that it allows Web services to be accessed behind a firewall, thus taking better advantage of the Wide Area Network (WAN) called the Internet. A SOAP message contains an eXtensible Markup Language (XML) document [18], where XML is used to describe the data to be exchanged. The Web services specifications are based on open standards, which makes it cheap for small or medium enterprises to implement them.

Although Web services can be used for EAI which enables business-to-business (B2B) interaction, there are some cases in which business software can not perform every task which is needed in order to achieve a business goal. Some tasks may still be performed by humans, despite the fact that the Web services technology enables interactions between business applications without humans getting in the way. The set of tasks which enables a business to achieve a business goal is called a *business process*.

In this thesis the term *workflow* will be used to represent the automation of a business process, in whole or part. A *prototype* will be developed that demonstrates the execution of a workflow, where the activities in the workflow are Web service invocations. Web services will be provided by different service providers. The workflow will consist of a set of activities which can by performed in sequence or in parallel.

Workflow enactment is the process of controlling the correct and reliable execution of activities by different processing entities [24]. A workflow enactment service is the core of a workflow management system (WfMS), because it handles the workflow execution. This enables flow independence, which means that the flow of activities in the workflow can be modified without changing the implementation of Web services. On the other hand, the implementation of Web services can be changed without having to modify the flow of activities in the workflow.

The benefit of using Web services to perform activities in a workflow is that the Web services technology is a standards-based realization of an service-oriented architecture (SOA) [19]. SOA aims at removing redundancy via reuse of services, which implies that Web services can be shared among different types of workflows. According to [12], there are two types of Web services:

- **Composite:** A *composite* Web service combines the functionality from other Web services.
- **Basic:** A *basic* Web service only accesses the local system.

A workflow enactment service can contain one or more workflow engines. A *centralized* approach to workflow execution, means that only one workflow engine, which is located on a single server, is responsible for doing workflow enactment. Web services technology enables workflows to span a great variety of resources. However, as workflows become more complex, there is a greater pressure on workflow enactment services to deliver an acceptable level of performance. This is especially true for a centralized workflow enactment service. A centralized approach to control the execution of a workflow is not an optimal solution, when considering the decentralized nature of Web services and the Internet. The problem is that a centralized workflow engine can easily become a communication and processing bottleneck [24]. A centralized workflow enactment service can also become a central point of failure [24].

One way of solving the issues associated with a centralized workflow enactment service has been to distribute parts of the workflow among several workflow engines [24]. This solution is also known as a decentralized approach to workflow enactment. A workflow engine that is able to communicate with other workflow engines will be referred to as an *agent*. This is because an agent can act as both a client and a server. However, the traditional approach to decentralized enactment of distributed workflows has been to pre-allocate resources prior to the workflow execution [24]. As a result, some resources are occupied when they are not actually being used. Therefore, this approach wastes resources, which is a downside, because the wasted resources could potentially have been used for other purposes.

The motivation for this thesis is to develop a prototype that demonstrates a decentralized approach to workflow enactment, where solutions to the issues associated with a centralized workflow enactment service and a traditional decentralized workflow enactment service will be tested and evaluated. More specifically, the decentralized approach will be based on a continuation-passing enactment style, where a *continuation* represents the rest of an execution during a workflow execution [24].

According to [24], some of the advantages of using a decentralized workflow enactment service, which is based on a continuation-passing mechanism, can be described as follows:

- **Scalability:** *Scalability* is the ability to execute a growing number of concurrent workflows, or instances of the same workflows, simultaneously. Processing bottlenecks can be avoided by adding more agents in the workflow enactment service, to accomodate growth with capacity.
- **Adaptability:** *Adaptability* is the ability to change, or be changed, to fit changed circumstances. There is no need for static analysis of the workflow prior to the workflow execution, because agents are able to adopt to changes by changing the way they enact a workflow at run-time. A centralized worklfow enactment service is also adaptable, because it analyzes the workflow dynamically at run-time. A traditional decentralized workflow enactment service is not adaptable, because it does a static analysis of the workflow prior to the workflow execution.
- **Robustness:** *Robustness* has to do with automatic recovery of partial failures. Therefore, no central point of failure exists. This means that if one agent for some reason is unable to execute a workflow, then the workflow execution can be assigned to another agent at run-time. In addition, availability can be increased by adding more agents in order to minimize the effects of partial failures.

The prototype will be developed in a top-down manner. This implies that the functionality of the prototype will be defined first. A business scenario will be chosen as a basis for the creation of meaningful workflows, because meaningful workflows will make it easier to demostrate how a continuation-passing enactment style works in practice. A meaningful piece of functionality can be specified as a use case, where a use case is a Unified Modeling Language (UML) notation [14]. The rest of this thesis is organized as follows:

- **Chapter 2:** *Chapter 2* explains the concepts which are relevant for this thesis.
- **Chapter 3:** *Chapter 3* specifies the goal for this thesis, and it describes the minimum requirements for the prototype.
- **Chapter 4:** *Chapter 4* explores different design artifacts which are based on the requirements.

- **Chapter 5:** *Chapter 5* describes in detail how to realize the design artifacts based on a specific application scenario.

- **Chapter 6:** *Chapter 6* describes how the prototype was tested. An evaluation is given based on the results from the tests of the prototype.

- **Chapter 7:** *Chapter 7* discusses what has been learned during the development process, and it suggests further improvements to the prototype.

Chapter 2

Background

This chapter presents some relevant background information.

2.1 XML

2.1.1 Definition

According to [18], the *Extensible Markup Language* (XML) can be defined as:

> *a metamarkup language for text documents that allows developers and writers to invent the elements they need as they need them, where an element is XML's basic unit of data and markup.*

This definition defines XML as a highly flexible format for data. The first version of XML came in 1998 and was called XML 1.0 [18]. It was based on Unicode 2.0, but it soon became outdated as Unicode evolved. Therefore, an updated version of XML called XML 1.1 was released in 2004 [18]. In addition, the strength of using XML is described by [18] in the following statement:

> *XML and Unicode guarantee that your data files will be portable across virtually every popular computer architecture and language combination in use today (2004).*

This statement states that XML is language and platform independent [18].

2.1.2 Document

XML documents are text documents, which are composed of characters. If an XML document is to take advantage of the features of XML 1.1, then it has to start with the tag $\langle ?xml\, version = "1.1"?\rangle$. XML 1.1 defines how an XML document should look like, what elements are legal, etc. In short, an XML document should only have one root element, because it usually represents the document itself. The root element can contain any number of elements. An element can include attributes which can be used to describe the data it contains.

2.1.3 Schema

An *XML schema* makes it possible to define a specific type of XML document. It allows for the creation of simple and complex types. Any XML document that does not follow the rules of the XML schema is not accepted as a well-formed XML document. One rule may be that the data contained in an element is of a specific type. For example, a string of letters is not accepted if a numeric type was expected. The XML document should always include information on where to locate an XML schema if it uses one.

2.1.4 Parser

XML parsers read the data contained in an XML document, and is usually done by following some rules defined in an XML schema. XML parser are required to support either UTF-8 or UTF-16 as input encoding formats [18], which are Unicode formats. According to [18], the application that receives data from the parser may be:

- **A web browser:** *A Web browser* that displays the document to a reader.
- **A word processor:** *A word processor* that loads the XML document for editing.
- **A database:** *A database* that loads the XML data in a new record.
- **A program written in Java:** *A program written in Java* that converts the XML data into a manageable representation.

2.1.5 Document Object Model (DOM)

The World Wide Web Consortium (W3C) [13] defines DOM as:

> *a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.*

According to [13], a DOM implementation is:

> *that piece of software which takes the parsed XML or HTML document and makes it available for processing via the DOM interfaces.*

However, most DOM implementations provide a simple parsing interface that accepts a reference to an XML document file, or stream [18].

Problem The downside of using DOM is that the entire document must be read and parsed, by a DOM parser, before it is available to a DOM application [18]. This implies that the entire document must be stored in memory before it can be manipulated, which can be a big disadvantage if the document is large.

2.2 Web service

In order to understand the Web services technology, it is important to have a basic knowledge of XML. In addition, it is important to know the benefits of XML, because XML forms the basis for making Web services language and platform independent.

2.2.1 Definition

According to [3], a Web service can be defined as:

> *a piece of business logic, located somewhere on the Internet, that is accessible through standard-based Internet protocols such as HTTP or SMTP.*

This is a fairly simple definition, but there exist other definitions of Web services which are longer and more specific [12]. The key to the success of the Web services technology is XML, because it provides a standard way to define the syntax of exchanged documents.

2.2.2 Standards

The Web services technology includes three open standards named SOAP, WSDL and UDDI, which will be described separately. In order for a Web service to be accessible, it should at least be based on the two standards named SOAP and WSDL.

SOAP

SOAP was originally an acronym for Simple Object Access Protocol, but the acronym expansion is no longer used [2]. SOAP itself is really just a thin wrapper around an XML application payload [5], and it can be transported across the network using HTTP, FTP and SMTP protocols. This means that SOAP is a quite flexible messaging service. Many programming languages has a built-in support for this protocol. A client, written in one programming language, can send SOAP messages to a server, written in another programming language. Services provided by a server through a SOAP interface are traditionally called Web services. The SOAP message consists of an envelope that contains a header and a body. The header element consists of header blocks that contains optional or mandatory information about how to process and route the SOAP message. The body element is always mandatory and contains the message content intended for the final receiver. SOAP can be used as a client-server model for making RPC method invocations, but messages can also be sent only in one direction to a receiver [19].

WSDL

WSDL stands for Web Service Description Language [19], and describes where and how to access a Web Service. It describes which messages go in and come out from a Web Service, and also its operations, in an abstract way. WSDL 1.1 has been standardized in 2001 by W3C as a protocol that logically describes the functionality of a Web Service and is constructed using the XML format. WSDL includes a support to describe Web Services that are constructed using a RPC style approach and Web Services that are constructed using a message style approach. WSDL tells a client what the Web Service does, how to use it and where it is offered. Some basic set of tags has to be present in a WSDL document so that a WSDL parser understands how to parse the document. A WSDL file starts by specifying the XML version and encoding style. The WSDL document starts with a definitions tag that contains all the other definitions under a single namespace. Some of the other mandatory tags are described here:

- **Message:** This tag describes messages, that go in and come out from a Web service, in an abstract way.
- **PortType:** This tag describes a set of operations, supported by a Web service, in an abstract way.
- **Operation:** This tag describes a single operation.
- **Binding:** This tag is a concrete protocol describing a particular portType, and may be used to specify a RPC style messaging approach using SOAP.
- **Port:** This tag must be defined to specify an address to a Web service and associate it with a binding tag.
- **Service:** This tag is a collection of port tags that can be used to specify multiple Web service addresses.
- **Documentation:** This tag can be used to write human readable words to describe how the Web service works.

UDDI

UDDI stands for Universal Description, Discovery and Integration [19] and is a registry of Web services. Categories separate Web services and it is possible to search for a specific Web service using the UDDI registry.

2.2.3 Composition

A *composite Web service* invokes one or more other Web services and combines their functionality. In contrast, a Web service that does not invoke other Web services is called a *basic Web service*. The process of developing a composite Web service is referred to as a *Web service composition* [12], but the details on how this is done is usually kept private, for example within a company.

2.2.4 Orchestration

A Web service composition may require that Web services invocations need to be coordinated. This is accomplished via *Web service orchestration*, which controls the order in which different Web services are invoked [12]. In addition, it controls the conditions under which Web services should or should not be invoked [12]. An *orchestrated Web service* invokes one or more other Web services from potentially multiple service providers in a coordinated fashion [1]. A Web service orchestration can be modelled using Petri nets [22] or a variety of UML activity diagrams [14]. These models can be used to analyze the orchestration.

2.3 Workflow

2.3.1 Related concepts

Processes

A process in the real world consists of a number of tasks to be to be executed, where a set of conditions determine the execution order [22]. A task is a logical unit of work. According to [22], a task can belong to each of the three categories:

- **Manual:** A task is performed by a person, without any intervention by an application.
- **Automatic:** A task is performed by an application, without any human intervention.
- **Semi-automatic:** A task involves the use of an interactive application, where the latter means that a person can interact with an application.

A generic name for person, application or a group of persons or applications is often referred to as a resource [22].

A directed graph can be used to visualize the flow of activities in a process, where an activity represents the performance of a task by a resource. For example, figure 2.1 shows a directed graph that illustrates the flow of five activities called *A*, *B*, *C*, *D* and *E*. The graph used in figure 2.1 gives a high-level view of the ordering of the steps within the process.

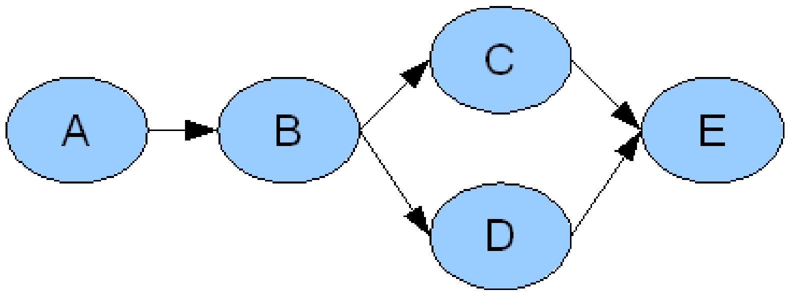


Figure 2.1: An example of a process

Some activities in figure 2.1 are dependent on the completion of other activities. For example, the arc from *A* to *B* illustrates that *A* is dependent of the completion of *B*.

Figure 2.1 also shows that C and D can be processed in parallel. Finally, E depends on the completion of both C and D.

Business processes If a process intends to achieve a business objective, then it can be referred to as a business process. Traditionally, business processes only involved manual activities, but the invention of information technology has made it possible to automate some of the activities.

The example in figure 2.2 illustrates how to handle an insurance claim, and it is a modification of the example used in figure 2.1. The only difference is that the activities, called A, B, C, D and E, are given meaningful names in order to get a better understanding on how to achieve the business objective.

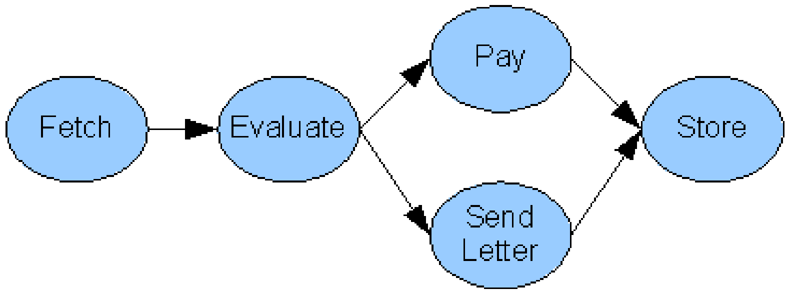


Figure 2.2: An example of a business process

The first activity *Fetch* means to fetch a new insurance claim. The next activity *Evaluate* means to evaluate the claim. The two activities *Pay* and *Send* means to pay money to the customer and to send documents to the customer respectively. The last activity *Store* means to store the results into a persistent storage.

2.3.2 Definition

The Workflow Management Coalition (WFMC) [4] defines a *workflow* as:

> *the computerized facilitation or automation of a business process, in whole or part.*

A distributed workflow is executed by multiple processing entities, where each processing entity only executes a portion of the workflow.

2.3.3 Specification

A workflow specification is a formal, executable description of a workflow [12], and it can be visualized using graphical notations. The purpose of the workflow specification is to automate the steps of a business process. A workflow specification can be expressed in Web Services Business Process Execution Language (WS-BPEL) [17]. A WS-BPEL specification can be modelled using Petri nets [22] or a variation of UML activity diagrams [14]. These models can be used to analyze a workflow by observing the order in which activities are executed, and under which conditions activities are executed.

It is important to understand the term *case*, when dealing with a workflow specification. A case can be described by using the example in figure 2.2, because it begins at the start of the first activity and ends after the completion of the last activity. A case can have different states during its lifetime. According to [22], the state of a case consists of three elements:

1. The values of the relevant case attributes.
2. The conditions that have been fulfilled.
3. The content of the case.

The term *work item* is also important to understand, when dealing with a workflow specification. It can be regarded as the combination of a case and a task which is just about to be carried out [22].

The relationship between a task, a work item and an activity is illustrated in figure 2.3. An arc from the task to the work item only means that a task becomes an work item, but only if the state of the case allows it. The latter is illustrated by an arc from the case to the work item. An arc from the work item to the activity means that the activity is the performance of the work item.

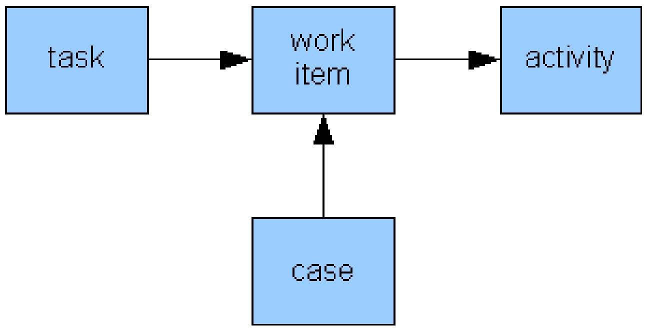


Figure 2.3: The relationship between a task, a work item and an activity

The examples in figure 2.1 and figure 2.2 only involve sequential and parallel flows, and all the nodes are of the same type. However, there are situations where a condition in a process allows the creation of an alternative flow. Therefore, in order to visualize a workflow specification using a directed graph, it is important to represent different types of nodes using separate notations. According to [12], there are three types of nodes:

- **Work node:** Work nodes represent work items to be performed by a resource.
- **Routing node:** Routing nodes define the order in which work items should be executed, and allow the definition of parallel and conditional activation of work nodes.
- **Start and completion nodes:** Start and completion nodes denote the starting and ending points of the workflow.

These types of nodes can be represented graphically is by using a variation of UML diagrams, but they can also be represented by using variations of Petri nets.

The example in figure 2.2 is extended, in figure 2.4, to include alternative flows in order to demonstrate the use of different types of nodes in a variation of the UML activity diagram [14]. Some of the notations, in figure 2.4, have been inspired by [12].

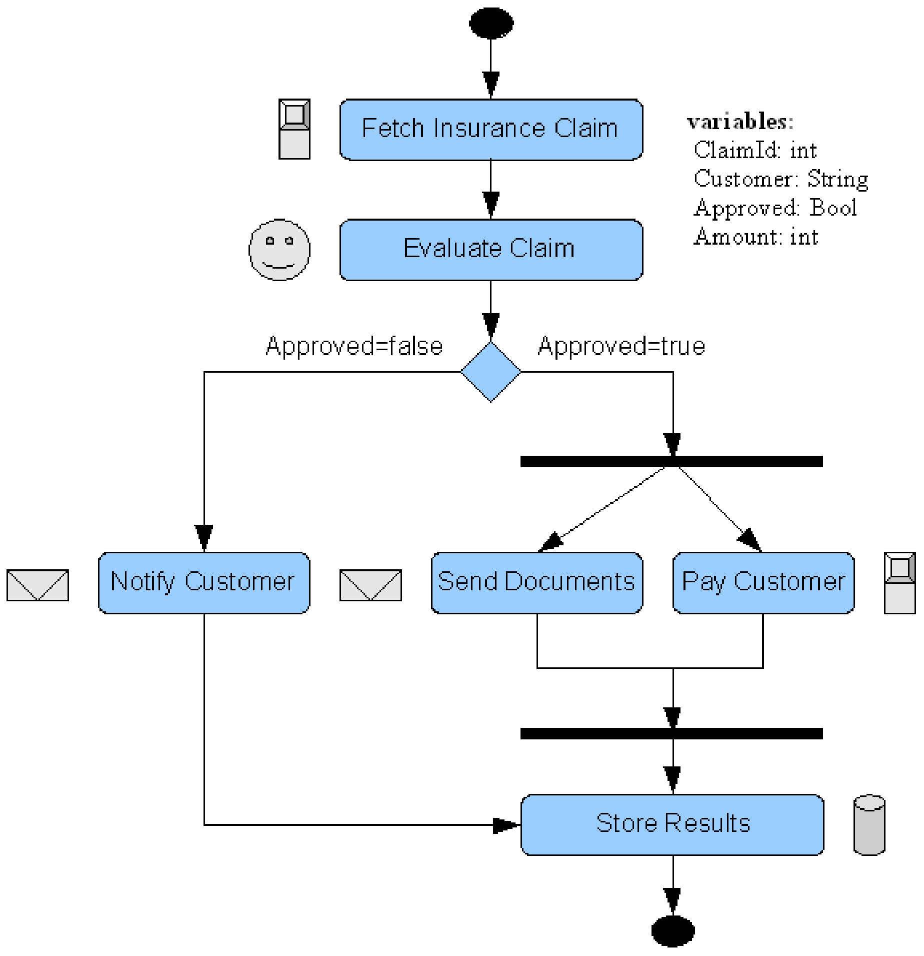


Figure 2.4: A workflow specification that models an insurance claim

Table 2.1 associates the different UML symbols with their respective node types.

Symbol	**Name**	**Node Type**
▢	Rounded box	Work node
◇	Diamond	Routing node
▬	Horizontal bar	Routing node
●	Filled circle	Start and completion node

Table 2.1: UML symbols and node types

The graphics next to the work nodes indicates what type of resource should perform the work item. An example of a resource can be a database, a human or an application. A set of variables are used to exchange data among work nodes, and they can also be used for routing purposes.

The example in figure 2.4 shows that an insurance claim can be rejected. If this happens, then the customer will not receive any payments. Instead, the customer will be notified via email with a description of what went wrong. The decision to reject or to approve an insurance claim is made by a human during evaluation, because this is indicated by the graphics next to the work node.

2.3.4 Workflow engine

A workflow engine executes workflow instances, where a workflow instance is an execution of the workflow [12].

2.3.5 Workflow execution

An general insight into the interactions of a workflow engine is given in figure 2.5.

The steps in figure 2.5 will be discussed briefly. Before the workflow engine can begin to work, it has to retrieve a workflow specification from a repository, as shown in *step two*. The workflow specification contains work nodes, routing nodes and start and completion nodes. The workflow engine knowns what type of node it is handling at any point of the workflow execution. If a node is a work node, then the workflow engine usually contacts a message broker, as shown in *step three*. The message broker selects a resource that belongs to the work item, and forwards this information to the workflow engine, as shown in *step four*. When the workflow engine knows which resources to use, it places the work to be done in the work queue of the selected resource, as shown in *step five*. The resource then pulls work, which could be a query to a database, from the work queue. Another form of interaction in *step five* is for the workflow engine to push work to the resource, meaning that the workflow engine invokes a method of the resource API. *Step one* illustrates that the workflow engine checks for the results of completed work items, before it executes the next node in the workflow specification.

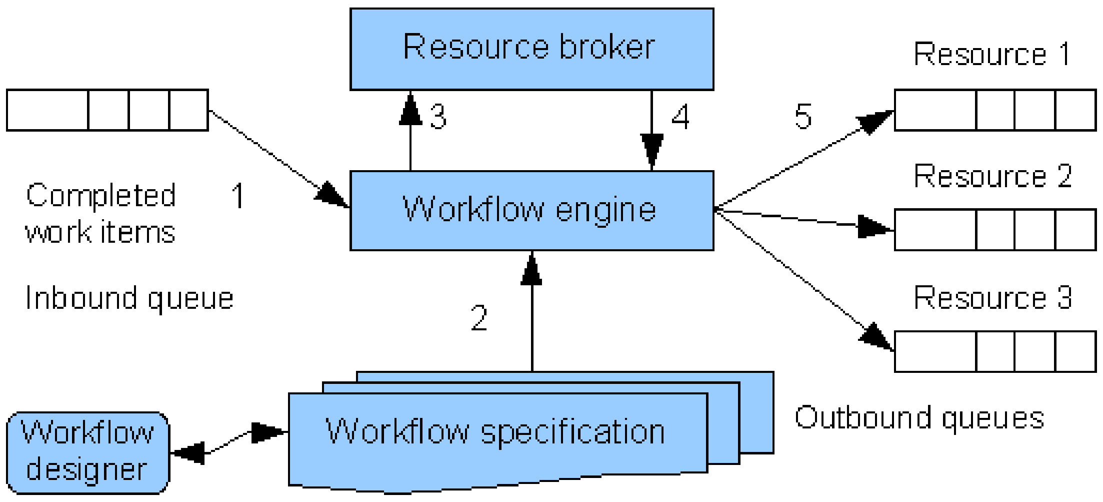


Figure 2.5: The general interactions of a workflow engine

2.3.6 Workflow enactment

Workflow enactment is defined as the process of controlling the correct and reliable execution of activities by different processing entities [24].

2.3.7 Relationships between terms

Table 2.2 shows the relationships between various terms which have been explained earlier.

Basic Web service	$\longleftrightarrow$	Task / activity
Composite Web service	$\longleftrightarrow$	Workflow / business process
Web service orchestration	$\longleftrightarrow$	Workflow enactment / execution

Table 2.2: Relationships between various terms

2.4 Workflow enactment services

2.4.1 Definition

A workflow enactment service can consist of one workflow engine, but it can also consist of multiple workflow engines. The former can be referred to as a centralized workflow engine, while the latter can be referred to as a decentralized workflow engine.

2.4.2 Centralized

A centralized workflow engine is located on a single server, and it is responsible for controlling all the activities in a workflow. One example that shows an execution of a workflow is illustrated in figure 2.6, where a centralized workflow engine executes a workflow specification that contains a sequence of three work items named A, B and C. The sequence is defined as seq(A, B, C).

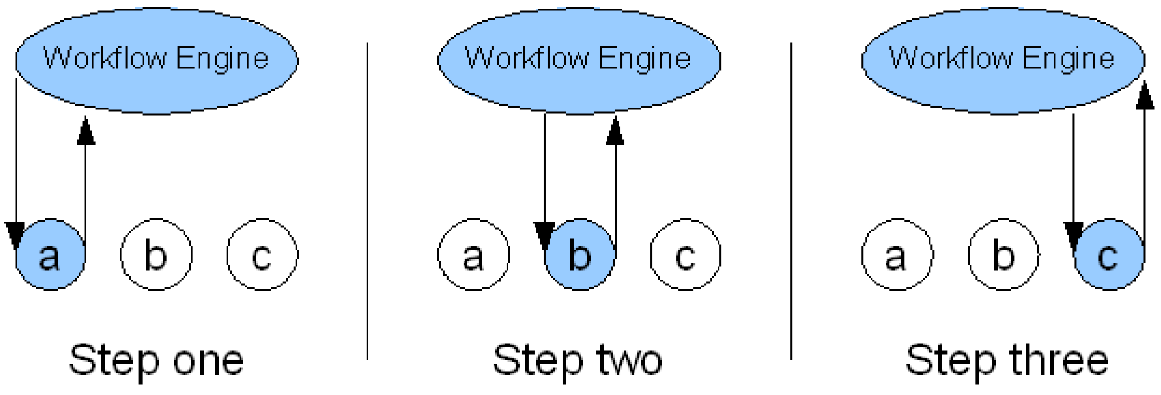


Figure 2.6: An example of centralized enactment

Figure 2.6 shows that the work items A, B, C are executed by resources located at the sites a, b and c, respectively. The arc from the workflow engine to a resource corresponds to step five in figure 2.5, and it illustrates a work item being sent from the workflow engine to a resource. If the resource has finished executing a work item, then it sends the results back to the workflow engine. *Step one*, *two* and *three* in figure 2.6 illustrates a synchronous form of interaction. For example, the workflow engine waits for the results from a, before it sends B to b.

Problem A problem is that centralized workflow engines can easily become a communication and processing bottleneck, and they can also become a central point of failure [24]. In addition, the centralized workflow engines are typically heavyweight. Therefore, the cost of deploying one is usually to high for a large number of small businesses [24].

2.4.3 Traditional decentralized

A traditional approach to decentralized workflow enactment requires pre-allocation of resources. In other words, the process is instantiated prior to its execution [24]. This explains why global coordination is needed. In addition, some pre-allocated resources can remain unused after the workflow execution has completed [24].

An example of a traditional decentralized enactment is shown in figure 2.7. The workflow specification that was illustrated in figure 2.6 will be used here too. The work items A,

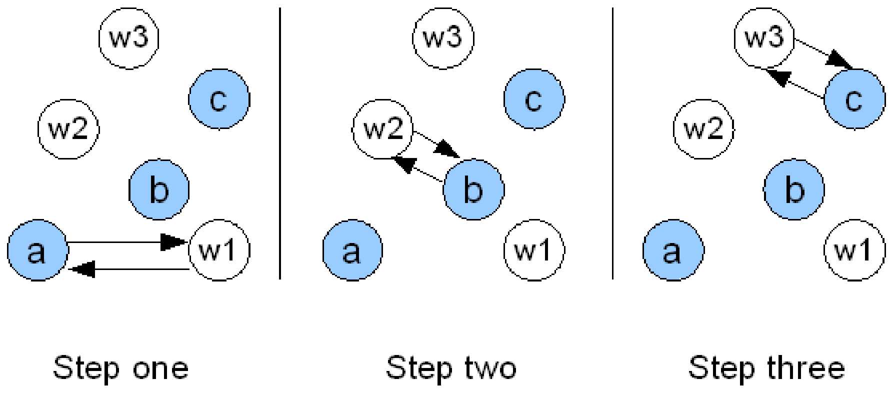


Figure 2.7: An example of traditional decentralized enactment

B and *C*, from the workflow specification, are executed at the sites *a*, *b* and *c*, respectively. Figure 2.7 shows that there are three workflow engines, located at sites *w1*, *w2* and *w3*, which are executing parts of the workflow specification. The arc from a workflow engines to a resource corresponds to step five in figure 2.5, and it illustrates a work item being sent from the workflow engine to a resource. In *step one*, one dedicated workflow engine, at site *w1*, is in charge of using the resource located at site *a*. In *step two*, another workflow engine, located at site *w2*, uses the resource located at site *b*. In *step three*, a third workflow engine, located at site *w3*, uses the resource located at site *c*.

All the resources which are needed during the workflow execution remain allocated in step one, two and three. The allocated resources are colored in blue.

Problem This form of enactment waste unused resources, which otherwise could have been used for other purposes. It is therefore an inefficient form of decentralized enactment. In addition, this form of enactment is not adaptable at run-ime.

2.4.4 Continuation-passing

Continuation-passing enactment is a decentralized approach to workflow enactment. A *continuation*, which represents the rest of an execution at a certain point of the execution, is passed from one agent to another using asynchronous messaging [24]. This approach does not need global coordination.

An example of a continuation-passing enactment is shown in figure 2.8. The workflow specification that was illustrated in figure 2.6 will be used here too. The work items *A*, *B* and *C*, from the workflow specification, are executed at the sites *a*, *b* and *c*, respectively. Figure 2.7 shows that there are three workflow engines, located at sites *w1*, *w2* and *w3*, which are executing parts of the workflow specification. The arc from a workflow engines to a resource corresponds to step five in figure 2.5, and it illustrates a work item being sent

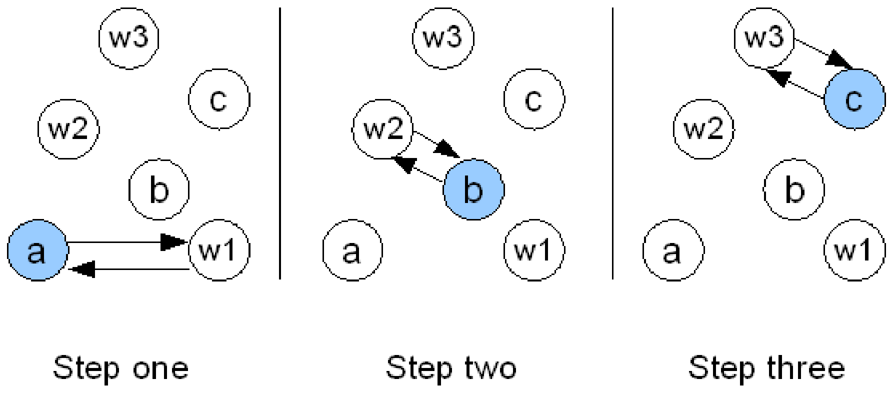


Figure 2.8: An example of continuation-passing enactment

from the workflow engine to a resource. However, all the resources are not pre-allocated prior to the workflow execution. In *step one*, a workflow engine located at site *w1* receives the whole workflow specification and uses the resource located at site *a* to execute the work item *A*. The continuation is the passed on to the next workflow engine located at site *w2*. In *step two*, the workflow engine located at site *w2* uses the resource located at site *b* to execute the work item *B*. Next, the continuation is passon on to the third workflow engine located at site *w3*. In *step three*, the workflow engine located at site *w3* uses the resource located at site *c* to execute the last work item *C*. The overall picture is that once a workflow engine has finished its work, then the continuation is passed on to the next workflow engine.

The benefit of using the continuation-passing mechanism is that no resources are pre-allocated. Only the resources which are actually needed are allocated in each step of the workflow execution. The allocated resources are colored in blue.

2.5 CEKK

2.5.1 A workflow model

The key abstraction in the *workflow model* is the term flow. The term *flow* corresponds to a workflow [24]. According to [24], a flow can be defined recursively as follows:

Flow ::== Empty Flow | Activity | Blackbox Flow
| **seq**(Flow*) | **fork**(join-agent, Flow*) | **or**(Flow*)
| **if**(Condition, Flow, Flow) | **loop**(Condition, Flow*)

As the definition shows, and according to [24], the flow includes three primitive flows as follows:

- **Empty Flow:** *Empty Flow* can be used to indicate the completion ot the execution of a flow.

- **Activity:** *Activity* is an activity that includes: an *agent* that is in charge of its execution, the program to be executed, a compensation program, the input and output data, etc.

- **Blackbox Flow:** *Blackbox Flow* is a flow whose internal structure is only known by its own agent, and it can be managed by a central workflow engine.

According to [24], some of the more complex flows can be described as follows:

- **seq:** *seq* defines a sequence of sub-flows.

- **fork:** $fork$ spawns multiple parallel branches of sub-flows. *joinagent* joins parallel branches, and it can be automatically chosen during the execution of a workflow.

- **or:** *or* enables execution from multiple alternative sub-flows.

- **if:** if chooses a flow by determining if a condition is true or false. A *condition* is defined on either flow-relevant data or execution status of the flow.

- **loop:** *loop* defines a sub-flow that is repeated until the condition is evaluated to be false.

An example [24] of a flow specification can look like the following:

$$\textbf{seq}(A_a,\ \textbf{fork}(\textbf{e},\ \textbf{or}(B_b,\ C_c),\ D_d),\ E_e)$$

A_a, B_b, C_c, D_d and E_e denote activities to be executed by agents at the sites a, b, c, d and e, respectively. A compensating activity for A_a would then look like A_a^{-1}. The flow specification can be represented in a tree structure, as figur 2.9 shows.

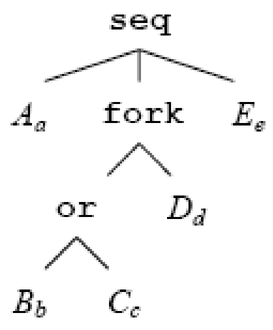


Figure 2.9: A tree structure of the flow specification[1]

[1]Copyright @ 2006 Yu and Yang [24]

2.5.2 Definition

CEKK is an abstract state machine for distributed and recoverable flow enactment, where C, E and K represent control, environment and continuation, respectively [24]. According to [24], the CEKK machine can be looked at from two perspectives:

- **Local CEKK machine:** A local CEKK machine defines possible states and their possible transitions locally at an agent.
- **Global CEKK machine:** A global CEKK machine defines the possible global states of a flow and the possible transitions among them. It consists of a number of local CEKK machines. The global state of the flow is the aggregation of the local states and the global state transitions are defined solely by the local state transitions..

A state of a local CEKK machine at agent p is a quadruple $< c, e, ks, kf >_p$, where c is called a control expression, e an environment, ks is called a success continuation and kf is called a failure continuation [24]. According to [24], they can be defined as follows:

- **control expression:** *control expression* represents the next flow to be enacted immediately.
- **environment:** *environment* is the runtime context of the flow.
- **success continuation:** *success continuation* is the continuation towards the successful end of the flow.
- **failure continuation:** *failure continuation* is the continuation towards the compensated end of the flow.

2.5.3 Transition rules

According to [24], the state transitions appears in one of four forms as follows:

1. **Local ongoing:** A local state transition performed by agent p.

   $$< c_0, e_0, ks_0, kf_0 >_p \rightarrow < c_1, e_1, ks_1, kf_1 >_p$$

2. **Remote forwarding:** Send state information from agent p to agent q.

   $$< c, e, ks, kf >_p \rightarrow < c, e, ks, kf >_q$$

3. **Local divergence:** Agent p spawns multiple parallel branches, where each branch is independent from the other branches.

   $$< c_0, e_0, ks_0, kf_0 >_p \rightarrow$$

   $$\{< c_1, e_1, ks_1, kf_1 >_p, < c_2, e_2, ks_2, kf_2 >_p, \ldots, < c_n, e_n, ks_n, kf_n >_p\}$$

4. **Local convergence:** Agent p joins spawned branches.

$$\{< c_1, e_1, ks_1, kf_1 >_p, < c_2, e_2, ks_2, kf_2 >_p, \ldots, < c_n, e_n, ks_n, kf_n >_p\} \rightarrow$$

$$< c_u, e_u, ks_u, kf_u >_p$$

Remote forwarding is asynchronous message passing between agents. In all other cases, state transitions are carried out locally at individual agents. This explains why global coordination is not needed among the agents [24].

The names of the transition rules have intuitive meaning. For example, the transition rules named A1 and S1 refers to an activity and a sequence, respectively. According to [24], the transition rules named *A1*, *A2*, *A3*, *S1* and *S2* can be listed as follows:

- **A1:** *A1* represents an agent q that is sending state information to another agent p.

  $$< A_p, e, ks, kf >_q \rightarrow < A_p, e, ks, kf >_p \quad if\ p \neq q$$

- **A2:** *A2* represents a successful execution of an activity at an agent p, where a compensation activity is added to the failure continuation.

  $$< A_p, e, ks, kf >_p \rightarrow$$

  $$< ks.head, succ\,(A_p) : e, ks.tail, A_p^{-1} : kf >_p \quad if\ succ(A_p)$$

- **A3:** *A3* represents a failed execution of an activity at an agent p, which means that the failure continuation is enacted.

  $$< A_p, e, ks, kf >_p \rightarrow < kf.head, fail\,(A_p) : e, kf.tail, kf >_p \quad if\ fail(A_p)$$

- **S1:** *S1* represents the last sub-flow to be enacted in a flow sequence, which means that nothing is pushed to the success continuation.

  $$< seq(fs), e, ks, kf >_p \rightarrow < fs.head, e, ks, kf >_p \quad if\ |fs| = 1$$

- **S2:** *S2* represents a flow sequence with multiple sub-flows, which means that the first sub-flow is enacted. The other sub-flows are pushed to the success continuation.

  $$< seq(fs), e, ks, kf >_p \rightarrow < fs.head, e, seq\,(fs.tail) : ks, kf >_p \quad if\ |fs| \neq 1$$

The next transition rules named *F1* and *J1* are somewhat complicated, therefore a state initially represented as $< c, e, ks, kf >$ will be shortened to $< c, ks >$. A reason for doing this is because recoverable flow enactment is not a goal for this thesis. The modified transition rules can be defined as follows:

- **F1:** *F1* represents agent p which spawns multiple parallel branches.

  $$< fork(q, f_1, f_2, \ldots, f_n), ks >_p \rightarrow$$

  $$\{< f_1, join\_succ : ks >_p, < f_2, join\_succ : ks >_p, \ldots, < f_n, join\_succ : ks >_p\}$$

where

$$join\_succ = join(q, and(succ(f_1), succ(f_2), \ldots, succ(f_n)))$$

- **J1:** *J1* represents joining spawned branches.

$$\{< join\_succ, ks >_q, \ldots, < join\_succ, ks >_q\} \rightarrow < ks.head, ks.tail >_q$$

The transition rules A1, A2, A3, S1 and S2 are valid for an abstract CEKK engine, while the transition rules F1 and J1 are valid for an abstract CK engine, where C stands for control and K stands for continuation.

Chapter 3

Requirements

3.1 Goal

A goal has been described in the task description for this thesis as follows:

> *The goal of this project is to design, implement and test a Java prototype that supports decentralized web services orchestration based on continuation-passing enactment of distributed workflows.*

Some of the related background information, in the previous chapter, give an explanation of the different terms used in this description.

3.2 Functional and data requirements

3.2.1 Functional requirements

The prototype have to include a Web application with a user interface, a workflow enactment service and at least two service providers. In addition, it is required to have an application scenario for the prototype which makes it possible to construct meaningful workflows. A meaningful workflow is important in order to better understand what is happening during the workflow execution. However, a specific application scenario will not be defined as part of the functional requirements. Any application scenario that fits the requirements is approved. The workflow must be an orchestration of Web services, where the Web services are provided by different service providers.

The functional requirements for the user interface, the Web application, the workflow enactment service and the service providers will be described separately.

Service providers Each service provider must be independent from one another. The service providers have to provide simple Web services which can be accessed via the Web. It must be able to dynamically configure each Web service while it is running on a server. In addition, each Web service must be able to:

- Send and receive SOAP messages.
- Provide a WSDL description of its services that can be accessed via the Web.

Web application The user interface must be embedded in the Web application. The user interface must be Web-based, which means that it must accept input and provide output in the form of Web pages. The Web pages should be displayed in a Web browser, and they must contain pure HTML code. A user that is using the Web browser should be able to choose among different options in the Web page. The Web page must then list a set of options that the user can choose from. The user must also be able to make a request from the Web page. The Web browser should act as a proxy on behalf of the user. The request must be sent to the Web application and contain all the options which the user has chosen. The user must be able to view the response in a Web browser. As a summary, the user interface must be able to:

- Send and receive Web pages via the Web.
- Process Web requests.

A workflow specification should be sent to the workflow enactment service when a user makes a request via a Web browser. The workflow specification must be based on which options the user has chosen. The results received from the workflow enactment service must be sent as a response to the request made by a user via a Web browser. In order to fulfill these requirements the Web application must be able to:

- Assign a unique reference number to each workflow specification.
- Keep a list of available service providers.
- Send and receive SOAP messages.
- Provide a WSDL description of its services that can be accessed via the Web.

Workflow enactment service The workflow enactment service must consist of multiple workflow engines. It must support decentralized web services orchestration based on continuation-passing enactment of distributed workflows. The continuation-passing mechanism must follow the transition rules of an abstract CK engine. In addition, each workflow engine must be able to:

- Receive a workflow specification.
- Execute the whole workflow specification or parts of the workflow specification.
- Invoke a Web service from a service provider.
- Communicate directly with one another.
- Send and receive SOAP messages.
- Provide a WSDL description of its services that can be accessed via the Web.

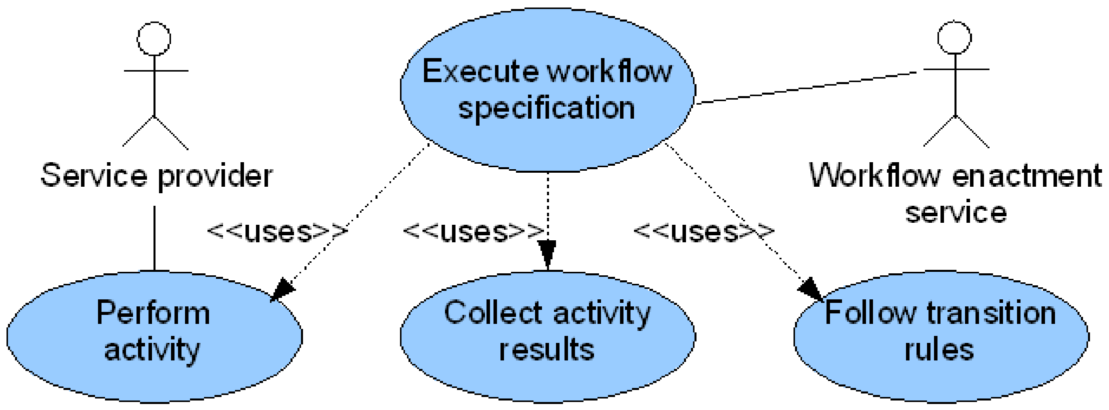


Figure 3.1: Workflow execution

The UML use case diagram [14], in figure 3.1, shows the requirements for the workflow execution.

In addition, the workflow enactment service must be able to execute a minimum of two independent parts of a workflow specification in parallel. The workflow enactment service must also be able to execute multiple workflow specifications concurrently.

3.2.2 Data requirements

The format of the workflow must at a minimum be based on the format that the CK machine uses. This means that it should be possible to identify the tag $< c, ks >$ within the structure of the workflow, where c stands for control and ks stands for success continuation.

Chapter 4

Design

This chapter will identify some of the design artifacts which need to be organized, while taking into consideration the requirements from the previous chapter. The design artifacts will be organized in a way so that they fulfill the requirements for the prototype.

4.1 Components

From the requirements it has been possible to identify three artifacts, namely a Web application, a workflow enactment service and a set of service providers. It is assumed that these artifacts are independent processes. Therefore, each artifact can be thought of as a component.

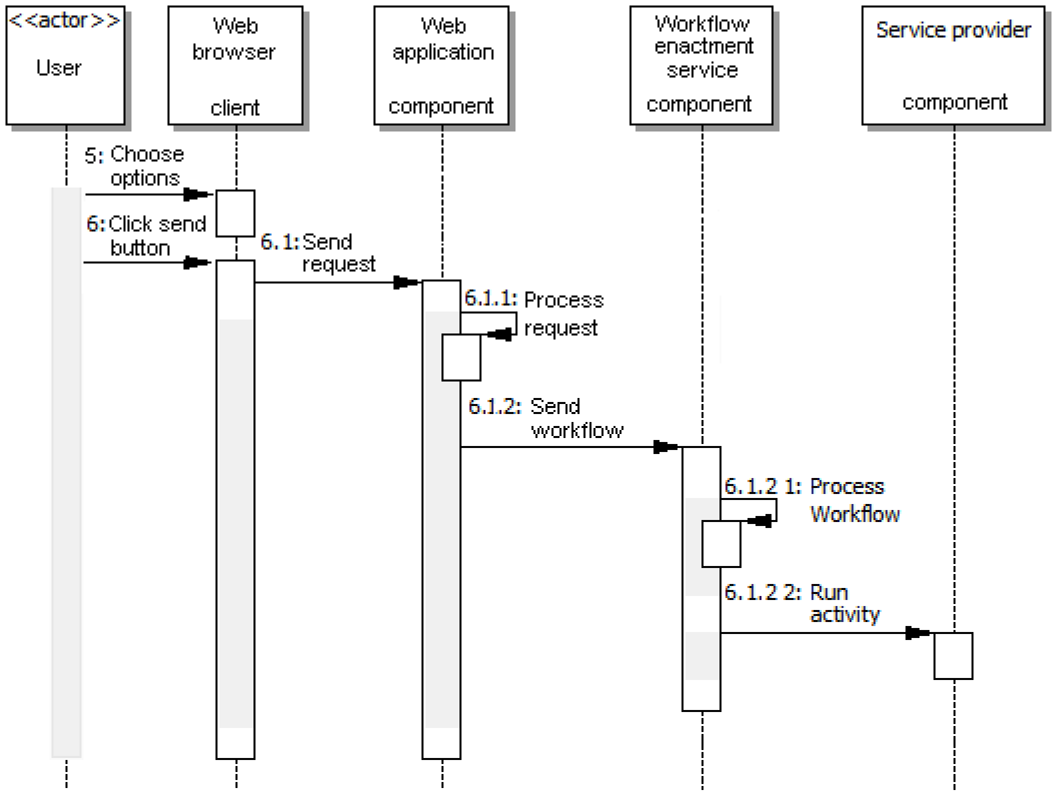


Figure 4.1: A view of the prototype components

A UML activity diagram in figure 4.1 is used to illustrate a minimum set of messages which must be sent between the different components. This is helpful in order to understand what form of communication is needed between each component. In addition, knowing what type of information is passed between each component gives an idea on what type of information must be handled by each component. Each component will be described separately.

4.1.1 Web application

The *Web application* component should allow users to interact with the prototype. Allowing this interaction to take place through a Web-based user interface means that the prototype can be used from any location on the Internet. The only thing a user needs in order to interact with the prototype is a Web browser. This allows a wide range of application scenarios to be chosen. At the minimum the user must be able to choose at least two activities from the Web page, where the activities must be performed by different service providers. This is important in order to allow the workflow enactment service to execute two independent parts of a workflow specification in parallel.

If figure 4.2 there are three entities, namely a client, a server and a workflow enactment service. The Web application resides on the server.

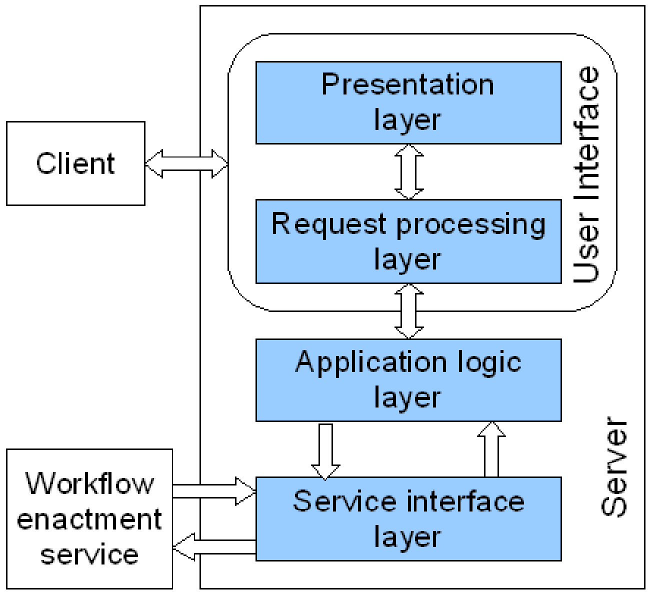


Figure 4.2: The architecture of a Web application

The *client* represents a Web browser, and displays Web pages to a user. The user interacts with the Web browser, while the Web browser interacts with the user interface of the Web application via the Internet.

The *server* can be separated into four layers, namely a presentation layer, a processing layer, an application logic layer and a service interface layer. Each layer can be described as follows:

- **Presentation layer:** The *presentation layer* is the static part of the user interface, and it determines what the Web pages should look like for the end user.
- **Request processing layer:** The *request processing layer* is the dynamic part of the user interface. This layer handles the requests from a Web browser and it replies to the Web browser by sending a Web page. The request processing layer can be a servlet, a Perl script, etc.
- **Application logic layer:** The *application logic layer* generates a workflow specification based on the data from the request processing layer. It also generates a unique reference number for each workflow specification. This layer also keeps an overview of the results received via the service interface layer.
- **Service interface layer:** The *service interface layer* is responsible for sending workflow specifications to the workflow enactment service via SOAP messaging. It is also responsible for receiving SOAP messages which contain the results from the workflow enactment service. It will also provide a WSDL description of its services, which acts as a service contract.

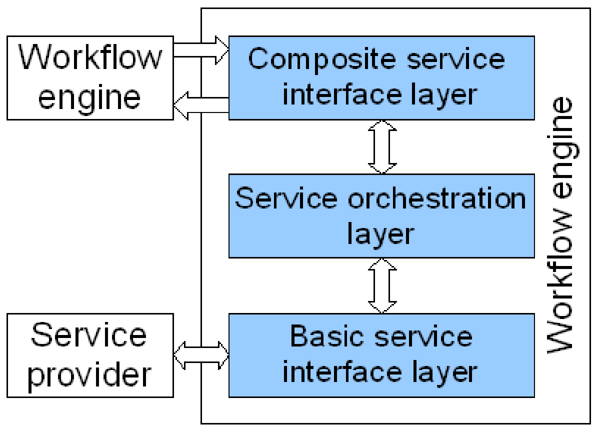
A template engine can be used to separate the presentation layer from the request processing layer. Web pages can be stored as templates, and represent the presentation layer. The templates can contain embedded scripting code. The template engine can generate HTML code from the templates. Using templates has a benefit because information in different languages can be stored in separate templates. The Web-based user interface represents both the presentation layer and the request processing layer.

4.1.2 Workflow enactment service

The *workflow enactment service* will consist of multiple workflow engines. The architecture of a workflow engine is shown in figure 4.3.

A workflow engine can be separated into three layers, namely an composite service layer, a service orchestration layer and a basic service interface layer. Each layer can be described as follows:

- **Composite service interface layer:** The *composite service interface layer* is responsible for receiving SOAP messages containing workflow specifications from the Web application or from other workflow engines. I should also be able to access other workflow engines via SOAP messaging. It should provide a WSDL description of its services, which acts as a service contract. As figure 4.3 tries to illustrate, the message passing between different workflow engines are carried out in an asynchronous manner. Therefore, once a message has been sent from a workflow engine, then no message is expected in return.



Workflow engine
Composite service interface layer
Service orchestration layer
Service provider
Basic service interface layer
Workflow engine

Figure 4.3: The architecture of a workflow engine

- **Service orchestration layer:** The *service orchestration layer* handles the processing and transformation of workflow specifications according to a set of rules which support decentralized Web services orchestration based on continuation-passing enactment of distributed workflows.
- **Basic service interface layer:** The *basic service interface layer* is responsible for accessing Web services offered by service providers in a synchronous manner.

4.1.3 Service providers

The *service providers* can provide a variety of different Web services. The architecture of a service provider is shown in figure 4.4.

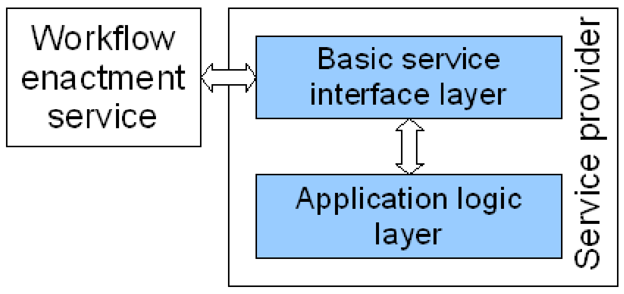


Figure 4.4: The architecture of a service provider

A service provider can be separated into two layers, namely an application logic layer and a basic service interface layer. Each layer can be described as follows:

- **Application logic layer:** The *application logic layer* depends on an application scenario. In general it will be responsible for manipulating application specific data. This layer should provide meaningful objects which can be manipulated.

- **Basic service interface layer:** The *basic service interface layer* is responsible handling SOAP messages in a request-response pattern. It will also provide a WSDL description of its services, which acts as a service contract.

Chapter 5

Implementation

This chapter will focus on how to realize the ideas presented in the previous chapter. The prototype will be implemented Java, as the goal for this thesis indicates. The Java programming language is very suitable because it provides an easy mechanism, which is called a package, that creates a namespace for each component of the prototype. Java byte code can also run anywhere a JVM is located.

5.1 Application scenario

The design components of the prototype will be realized to create an application scenario as follows:

- **Web application:** The *Web application* component will represent a travel agency that enables users, or customers, to book a trip via a Web browser. The trip can consist of a flight, a hotel and a car reservation.
- **Workflow enactment service:** The *workflow enactment service* component will consist of multiple workflow engines, where a workflow engine will be able to communicate directly with other workflow engines. This type of workflow engine will be referred to as a software agent, because an agent can act as both a client and a server. The software agents coexist in a distributed environment.
- **Service provider:** The *service provider* component includes multiple service providers. This is because the process of booking a trip can make use of a different service provider for each type of reservation. The service providers perform the actual flight, hotel and car reservation.

UML use case diagrams are being used in order to give a high-level view of the application scenario.

5.1.1 Actors

Customer The *customer* actor represents a human who is interested in booking a trip that the travel agency has to offer.

Flight The flight actor represents a fictional Web service provider that offers the possibility of booking a flight through its Web service interface. It also offers an opportunity to cancel a reservation.

Hotel The hotel actor represents a fictional Web service provider that offers the possibility of booking a hotelroom through its Web service interface. It also offers an opportunity to cancel a reservation.

Car The car actor represents a fictional Web service provider that offers the possibility of renting a car through its Web service interface. It also offers an opportunity to cancel a reservation.

5.1.2 Use cases for booking a trip

Figure 5.1 shows a UML use case for booking a trip. It shows a customer that can select the appropriate flight, hotel and car. Selecting a flight is always mandatory when booking a trip, but selecting a hotel or a car is an optional choice. It is assumed that the customer is already registered.

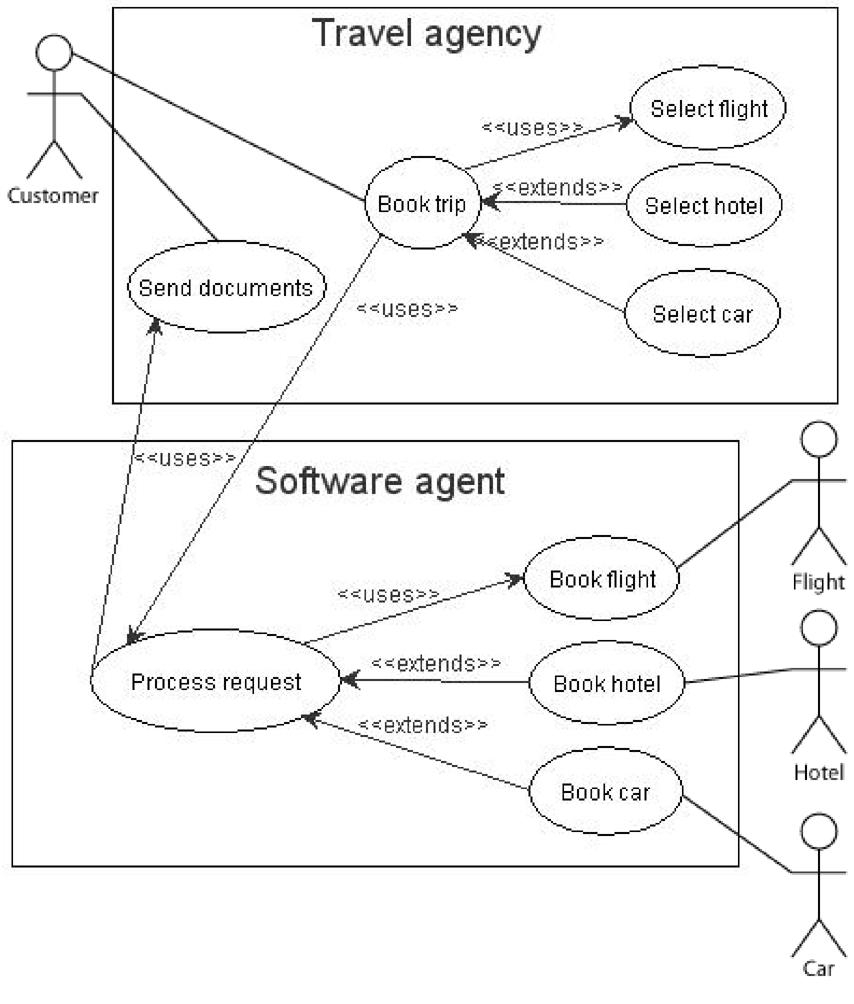


Figure 5.1: The booking of a trip by means of a UML use case diagram

5.2 Travel agency

The travel agency component is written in Java. The component is a simple Java class. This is shown in figure 5.2.

<<file>>
TravelAgency.class

Figure 5.2: Travel agency component

5.2.1 User interface

The Web pages are the static part of the user interface, and they are stored as templates on the server. These templates can be seen in figure 5.3. The static part of the user interface can be referred to as the presentation layer.

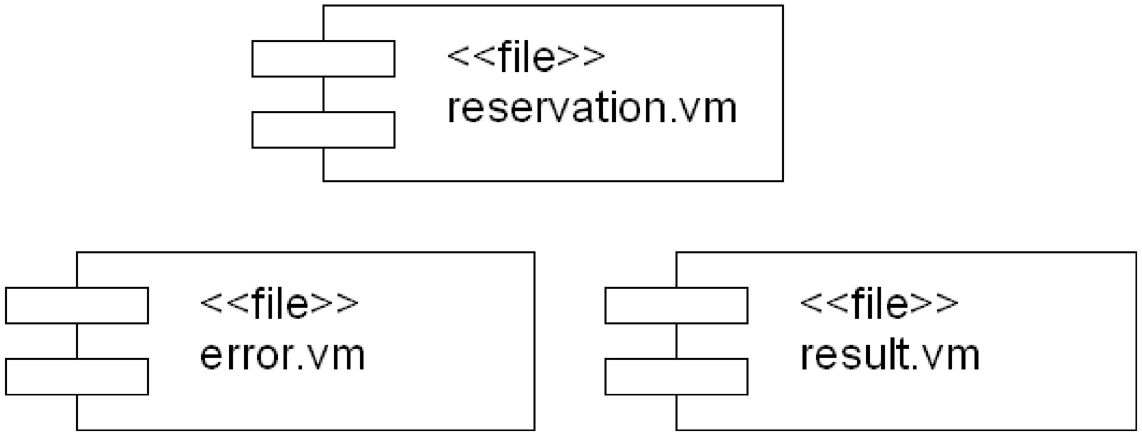


Figure 5.3: User interface components

A customer can choose a set of options from a Web page via a Web browser. Next, the customer can send a request by clicking on a button on the Web page. This Web page is generated from a template called *reservation.vm*. The Web page returned as a response to the request can be generated from a template called *result.vm* or *error.vm*. The template *result.vm* is used if the results have been successfully received from an agent. The template called *error.vm* is used if the time spent waiting for a result causes a timeout.

The dynamic part of the user interface accepts inputs from a Web browser. The inputs are handled by a servlet [16]. The servlet is also responsible for sending a response to the browser. The dynamic part of the user interface takes place at the request processing layer.

5.2.2 Application logic

Figure 5.4 shows how the use case *Book trip* in figure 5.1 is handled by the travel agency after the request processing layer has received a request from a Web browser, and it illustrates, at the application logic layer, how different workflow specifications are created based on what options the customer has chosen. In this example, the travel agency has a list of three software agents named *A*, *B* and *C*. Agent *A*, *B* and *C* are responsible for making different service providers perform the booking of a flight, hotel and car, respectively.

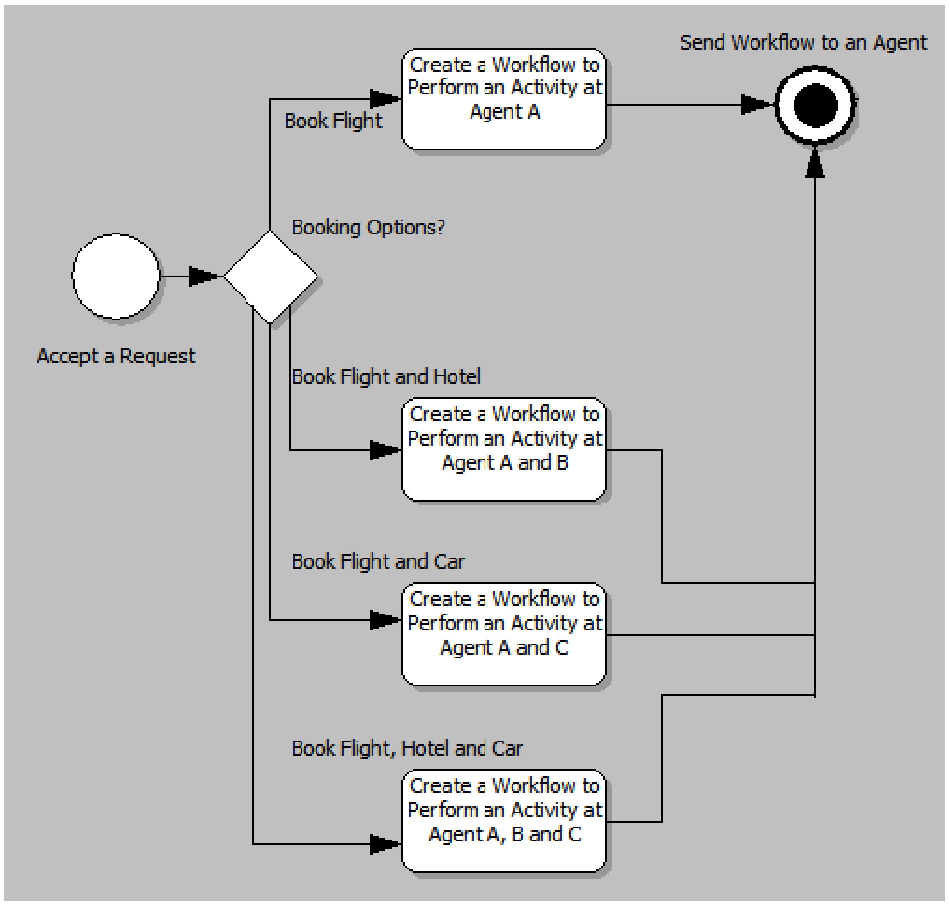

Send Workflow to an Agent
Create a Workflow to Perform an Activity at Agent A
Book Flight
Booking Options?
Accept a Request
Book Flight and Hotel
Create a Workflow to Perform an Activity at Agent A and B
Book Flight and Car
Create a Workflow to Perform an Activity at Agent A and C
Book Flight, Hotel and Car
Create a Workflow to Perform an Activity at Agent A, B and C

Figure 5.4: A flow diagram showing the creation of a workflow specification

From figure 5.4 it is possible to identify four different workflow specifications. An detailed example of a workflow specification will not be given until the continuation-passing mechanism which is used by the software agents has been explored in more detail. However, the workflow reference number is created dynamically for each workflow specification. In addition, the number of concurrently executing workflows which need to be joined is also created dynamically.

5.2.3 Resource access

The complete workflow specification is given to the resource access layer which sends the workflow specification to a predefined software agent.

5.2.4 Service interface

The service interface layer receives a messages which contain the results from completed workflow executions, which eventually ends up in a Web browser.

The use case *Send documents* in figure 5.1 only means that the results from the booking request should be displayed in the Web browser to the customer.

5.3 Software agents

The software agents are written in Java. Each software agent component is a simple Java class. This is shown in figure 5.5.

<<file>>
Agent.class

Figure 5.5: Software agent component

Each software agent contains a workflow engine. The software agents are able to communicate with each other in an asynchronous manner. The internal architecture of a software agent is shown in figure 5.6.

Before the use case *Process request* in figure 5.1 is explained, it is important to investigate some of the possible workflow patterns in more detail.

5.3.1 Abstract workflow patterns

The abstract workflow patterns has been inspired by the application scenario in figure 5.1. Four abstract workflow patterns illustrate different flows of four activities A, B, C and D, where A, B, and C represents to book a flight, hotel and car, respectively. Activity D represents an action which uses remote forwarding to send results to another agent. The abstract workflow patterns can be listed as follows:

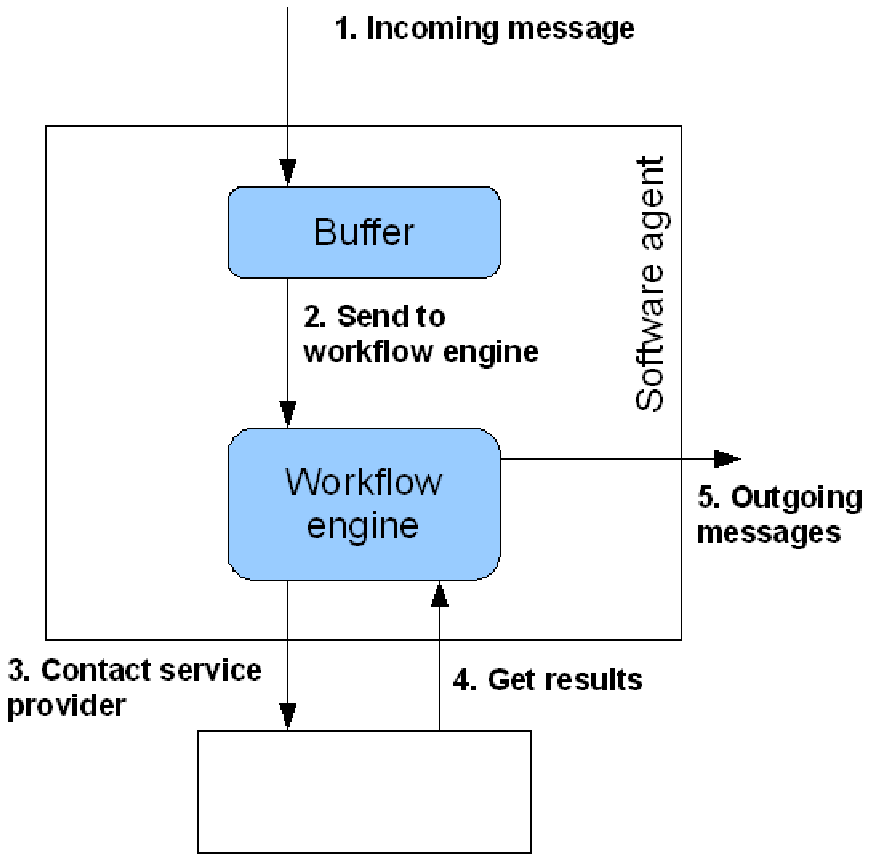


Figure 5.6: The internal architecture of a software agent

1. Sequence(Fork(A), D)
2. Sequence(Fork(A ,B), D)
3. Sequence(Fork(A ,C), D)
4. Sequence(Fork(A ,B ,C), D)

The fourth workflow pattern can easily be represented as a tree structure, with the *Sequence* element at the root, as figure 5.7 shows.

A running example

A running example, which is illustrated in figure 5.8, shows the message passing between agents, when using the workflow described in figure 5.7.

The agents are called start, a, b, c, d and e. The start agent receives a workflow and carries out the first step from the sequence flow in figure 5.7, which in this case is a *Fork*

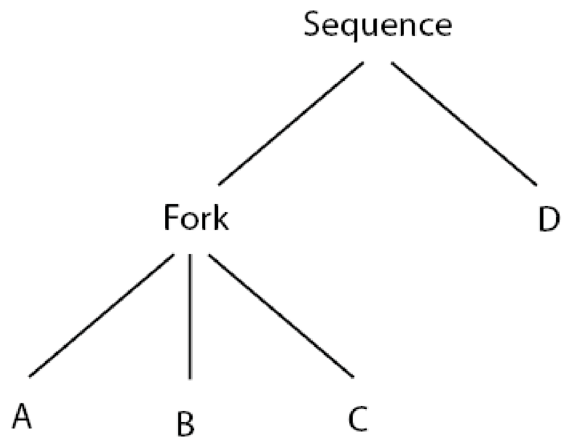


Figure 5.7: A tree structure representing a workflow specification

node. Figure 5.8 shows how three agents are run in parallel and later joined by agent e. The results from agent a, b and c are forwarded to agent d. This example shows a workflow that is successfully executed, but it only describes one specific workflow pattern.

5.3.2 Realization of abstract workflow patterns

The abstract workflow patterns will be realized by using XML to create the workflow specifications. However, before a workflow specification can be created using XML, there must exist some basic building blocks. These building blocks are the elements which must be defined in an XML schema. The abstract workflow patterns, combined with the application scenario, provide an helpful indicator towards which elements must be defined in the XML schema.

An XML schema for specifying workflow elements

A set of elements, which will be used in the worfklow specifications, can be defined in an XML schema as shown in Appendix A. The set of elements that is used in the XML schema is inspired by visualizing the abstract workflow patterns. The names of some of the elements are inspired by [24]. These few elements are sufficient in order to translate the abstract workflow patterns into executable workflow specifications. Some of the elements have attributes. Values can be assigned to the attributes dynamically during workflow execution.

The XML schema defines the root element of a workflow specification as $< cekk >$. The reason for choosing the name *cekk* is that the prototype will support recoverable distributed workflows. This property is not a part of the goal for this thesis, but an effort will be made to try and realize it anyway. The $< cekk >$ element has an attribute called *epr* whose value corresponds to the address of an agent which is currently executing the workflow specification. The $< cekk >$ element also has an attribute called $< id >$ whose value corresponds to the workflow reference number of the a workflow specification. The

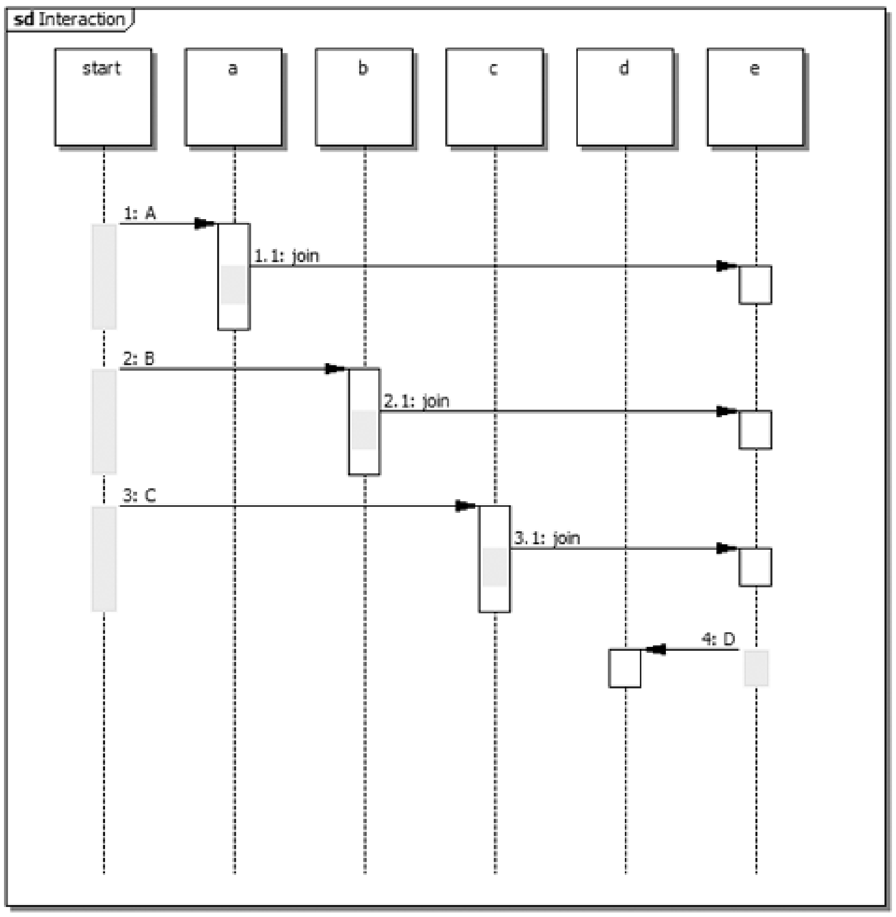


Figure 5.8: Agents running workflow pattern 4

elements that the $< cekk >$ element contains correspond to the elements of the quadruple on page 19, which describes the state of a local CEKK machine. Therefore, the $< cekk >$ element contains the elements $< c >$, $< e >$, $< ks >$ and $< kf >$. These elements can only contain one root element called $< seq >$, where the latter can contain the elements which are described as follows:

- $< activity >$: The $< activity >$ element is used by an agent to let a service provider perform an activity. The address of a service provider is stored in an attribute called *epr*. Every activity is defined by using the format of an $< activity >$ element, which consists of the elements $< method >$, $< arg1 >$, $< arg2 >$ and $< compensation >$. The $< method >$ element contains the name of a service provided by a service provider. The elements $< arg1 >$ and $< arg2 >$ contain data values which the service accepts. A $< compensation >$ element specifies the name of a service which reverse the changes made by a completed activity.

- $< fork >$: The $< fork >$ element creates parallel branches, where each branch contains a root element called $< cekk >$. The *epr* attribute in the $< cekk >$ element determines which agent should execute it. The $fork_epr$ attribute in the $< cekk >$ element specifies the address of the agent that executed the $< fork >$ element.

- $< end\_fork >$: Each branch created by the $< fork >$ element contains a $<$ $end\_fork >$ element at the end. The latter contains an attribute called *epr* which specifies the address of an agent that executes a $< join >$ element.
- $< join >$: The $< join >$ element has to attributes called *no* and $fork\_epr$, where the value of the former attribute is dynamically specified. The *no* attribute represents how many branches which should be joined. The $fork\_epr$ attribute specifies the address of the agent that executed the $< fork >$ element. This is useful in case a compensation is needed.
- $< remote\_trans >$: The $< remote\_trans >$ element is used to forward the $< cekk >$ element at the root to another agent.
- $< end\_trans >$: The $< end\_trans >$ element ends the workflow execution. It contains an attribute called *epr* which specifices the address of where the results from the workflow execution should be sent.

The elements which can contain a $< seq >$ element can be described as follows:

- $< c >$: This element is initially empty. However, before the first element in a $< seq >$ element can be executed, it has to be moved from the $< seq >$ element in the $< ks >$ element into the $< seq >$ element in the $< c >$ element. When this element has done its purpose, then it is removed. Next, the first element from the $< seq >$ element in the $< ks >$ element has to be moved into the $< seq >$ element in the $< c >$ element, and so on. Therefore, the $< seq >$ element in the $< c >$ element can only contain one element at any time, that is about to be executed.
- $< e >$: This element contains the results from successful or failed activities. Therefore, it is not restricted to only contain a $< seq >$ element, because it can contain arbitrary elements created on the fly during a workflow execution. The arbitrary elements then contain the results from successful or failed activities.
- $< ks >$: This element initially contains the whole workflow specification, and it represents the path towards a successful workflow execution. When the first element of a $< seq >$ element in the $< ks >$ element is removed, then the rest of the workflow specification remains in the $< ks >$ element. Therefore, a continuation is stored in this element.
- $< kf >$: This element is initially empty, but as activities completes successfully, then this element will contain a sequence of elements which can undo any changes made by successfully completed activities. Therefore, this element represents the path towards a failed workflow execution.

A valid workflow specification

A valid workflow specification has been created using a set of elements which the XML schema in Appendix A defines as legal. The abstract workflow pattern *Sequence(Fork(A ,B ,C), D)* can be realized by using the workflow specification in Appendix B. This workflow specification also contains sample values which correspond to a set of options which has been chosen by a customer. Figure 5.9 gives a graphical view of the workflow specification.

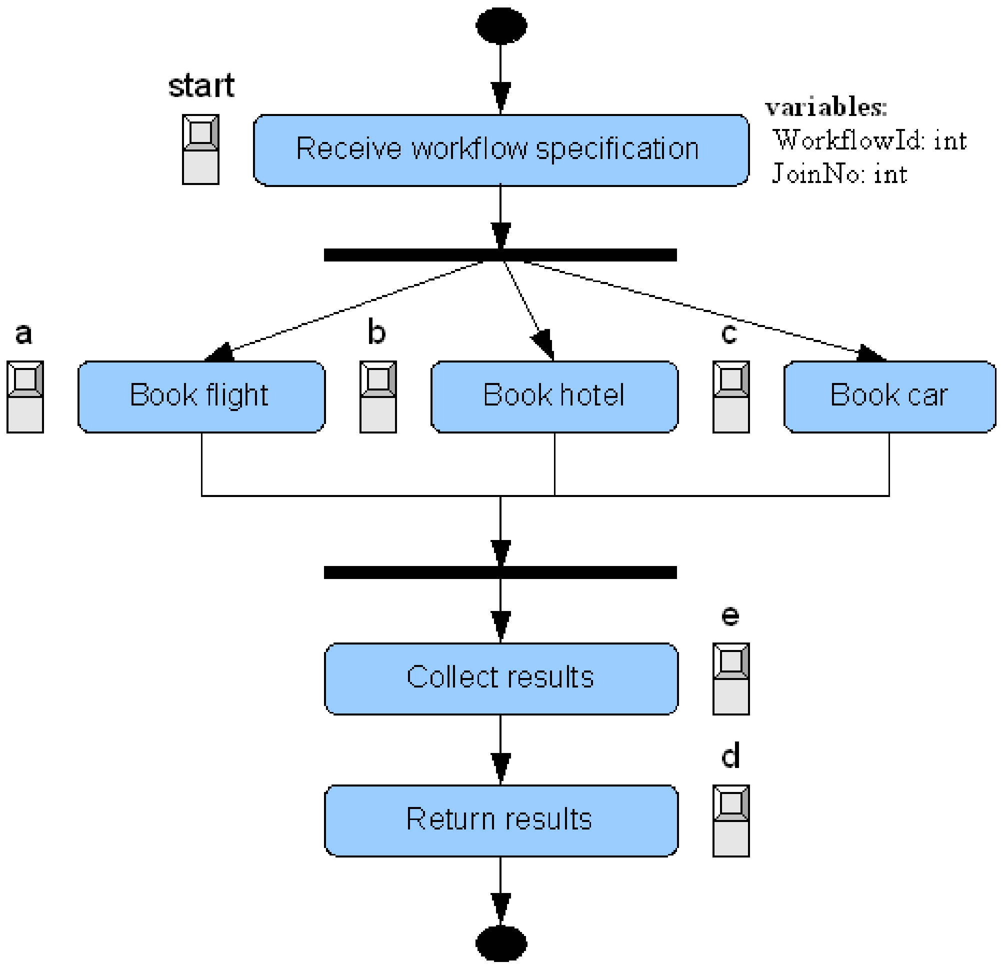


Figure 5.9: Agents running the workflow specification in Appendix B

An explanation of the graphical notations used in figure 5.9 can be found in table 2.1. However, the XML schema elements can be categorized into different node types, as figure 5.1 shows.

The workflow specification has to be extended into a CEKK specification, as shown in Appendix C, before it can be enacted by the software agents.

Work node	**Routing node**	**Start and completion nodes**
$< activity >$	$< fork >$	$< end\_trans >$
	$< end\_fork >$	
	$< join >$	
	$< remote\_trans >$	

Table 5.1: XML schema elements and node types

5.3.3 CEKK transition rules

The CEKK transition rules [24] will not be followed strictly, but they will be used as an inspiration for how the software agents execute a workflow specification. A description on how strictly the software agents follow the CEKK transition rules can be made by listing the following transition rules:

- **A1:** Rule *A1* corresponds to the $< remote\_trans >$ element in the XML schema. The software agents follow this rule strictly.
- **A2:** Rule *A2* is followed strictly by the software agents. The $< e >$ element in the workflow specification contains the result from a successfully completed activity.
- **A3:** Rule *A3* is not followed strictly by the software agents. The $< e >$ element in the workflow specification contains the result from a failed activity, but the failure continuation is not enacted until a software agent, that is performing a join, discovers the failed activity. Only then is the failure continuation enacted.
- **S1:** Rule *S1* is not followed strictly by the software agents, because the rule suggests that the workflow specification should be present in the $< c >$ element. However, the software agents expect the workflow specification to be present in the $< ks >$ element. If the workflow specification only contains one element in the $< seq >$ element, then this element will be pushed from the $< ks >$ element to the $< c >$ element.
- **S2:** Rule *S2* is similar to rule *S1*, but it suggests that the reminder of the workflow specification should be pushed from the $< c >$ element to the $< ks >$ element. This happens if the workflow has more the one element in the $< seq >$ element. However, a software agent only pushes the first element in the $< seq >$ element from the $< ks >$ element to the $< c >$ element. So the behaviour is fairly opposite to the transition rule.

A description on how strictly the software agents follow the CK transition rules can be made by listing the following transition rules:

- **F1:** Rule *F1* does not handle failure continuation, therefore this rule is not followed strictly by the software agents. The only similarity is that multiple parallel branches can be spawned.

- **J1:** Rule *J1* does not handle failure continuation, therefore this rule is not followed strictly by the software agents. The only similarity is that multiple parallel branches can be merged. The software agent performing the $< join >$ element will create a failure continuation that contains a $< fork >$ element.

5.4 Service providers

The service providers are written in Java. Each service provider component is a simple Java class. This is shown in figure 5.10.

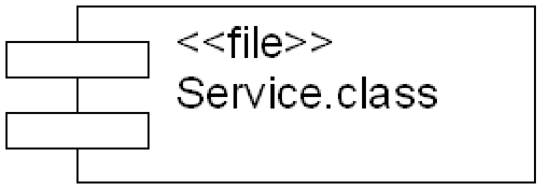


Figure 5.10: Service provider component

The service providers provide Web services which can be configured manually through a Web browser. Each Web service has several methods which manipulate the application logic data. The method for making a reservation has the same number of arguments in every Web service that is provided by the service providers. The service providers performs the actual activities in a workflow specification.

5.5 Development tools

5.5.1 Java Development Kit (JDK)

JDK [16], version 1.6, from Sun Microsystems was used to compile the Java code.

5.5.2 Eclipse Sofware Development Kit (Eclipse SDK)

Eclipse SDK [20], version 3.3.1, was used as a Java code editor, because it automatically detects syntax errors in the Java code before the code is compiled. It also gives valuable suggestions on how to correct the syntax errors. This helps in order to speed up the development process.

5.5.3 Apache Tomcat

Apache Tomcat [10], version 6.0, was used as an application server. A prototype component can be deployed in Apache Tomcat if has been packaged into a Web Application aRchive (WAR) file. If the prototype component has been successfully deployed, then Apache

Tomcat will handle HTTP invocations of the prototype component. It will also handle server-side dynamic content.

5.5.4 Apache Ant

Apache Ant [8], version 1.7, was used to automatically build and package the travel agency component, together with its dependencies, into a WAR file.

5.5.5 Apahe Velocity

The request processing layer is separated from the presentation layer with the help of the Apache Velocity Engine [11], version 1.4. The Apache Velocity Engine converts templates into HTML pages.

5.5.6 Apache Axis2

Apache Axis2 [9], version 1.3, was used as a core engine for Web services. It can be deployed into Apache Tomcat as a WAR file to handle the complexity in SOAP messaging, but it has not been tested in other application servers. Apache Axis2 is important because every prototype component must be able to send and receive SOAP messages. Apache Axis2 includes an object model called the AXIs Object Model (AXIOM) [9], which can transform the XML content of a SOAP message into an object model representation, which can easily be manipulated inside a prototype component. The object model can then be translated back into XML, using AXIOM, and sent via SOAP to a recipient. Apache Axis2 also automatically generates a WSDL file for every service, which can be viewed in a Web browser.

5.5.7 Apache TCPMon

Apache TCPMon [7], version 1.0, was used as a debug tool. This tool displays the contents of SOAP messages which are being sent between the components of the prototype.

5.6 Deployment

Each component of the prototype is contained within a Java package in order to give them a unique namespace.

5.6.1 Travel agency

The travel agency will be deployed to the application server as a WAR file. The WAR file contains all the necessary dependencies and configuration files which are needed in order to deploy the travel agency component. It also contains the component itself. One of these dependencies is the Apache Axis2 libraries which handle SOAP messaging. Two configuration files called *web.xml* and *services.xml* are needed. The former is needed by

Apache Tomcat, while the latter is needed by Apache Axis2. The *web.xml* configuration file describes how to access the travel agency component through a Web browser, while the *services.xml* describes how to access the travel agency component via SOAP messaging. A message exchange pattern (MEP) [9] can be specified in the *services.xml* configuration file, which in this case is both an In-Out MEP and an In-Only MEP.

5.6.2 Software agents

A software agent component will be uploaded as a service from the Apache Axis2 administration module, but before this happens it must be packaged into a file with an *aar* extension. The *aar* format is exactly the same as a *jar* [15] format, except that the names are different. The *aar* file must contain a ”service.xml” configuration file.

5.6.3 Service providers

The service providers are deployed in exactly the same way as the software agents.

Chapter 6

Testing

This chapter describes how the prototype was tested. It also describes the results from the various tests.

6.1 Configuration

Figure 6.1 shows an application server that contains different components. The location of the application server is *http://localhost:8080*. The components are named Hotel Service, Car Service, Flight Service, start, a, b, c, d, e and Travel Agency. All these components are independent from each other, and can live on separate application servers.

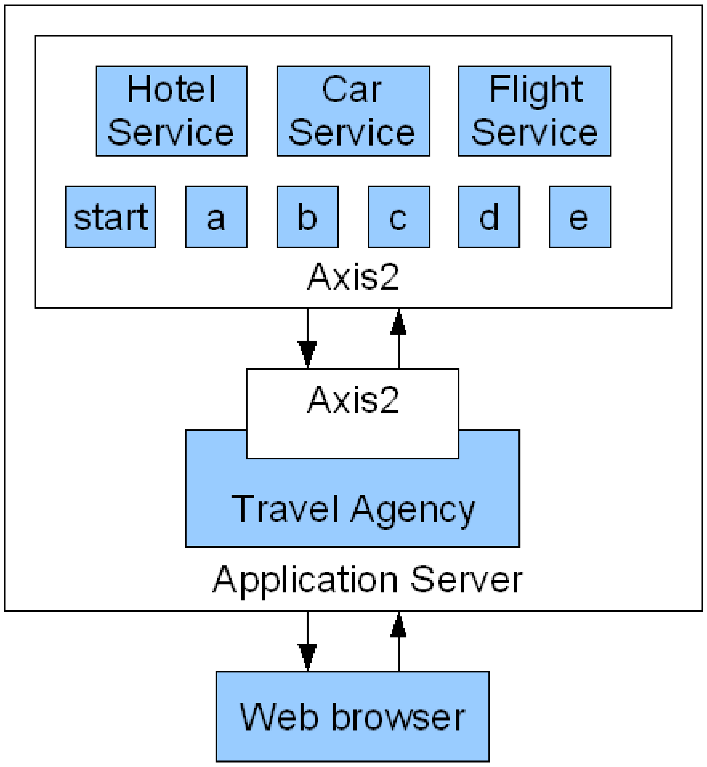


Figure 6.1: Test configuration

It can be seen that the travel agency component uses a separate Axis2 engine compared to the other components, but it is still able to communicate with the software agents called start, a, b, c, d and e. The software agents inside the Axis2 engine can communicate with each other, but they can also communicate with the service providers called hotel service, car service and flight service.

A Web browser can be used to access the travel agency component, but it can also be used to configure the service providers. Initially the service providers contains no application specific data. Therefore, the service providers have to be configured properly, before a test can start, as follows:

- **Flight Service:** The *Flight Service* is configured by typing the following url into a Web browser:

  http://localhost:8080/axis2/services/airline/
  setAvailableSeats?inFlightCode=AA0004&availableSeats=5

  This command will create a flight called AA0004 with five available seats. A list of commands for the *Flight Service* can be found in Appendix D.

- **Hotel Service:** The *Hotel Service* is configured by typing the following url into a Web browser:

  http://localhost:8080/axis2/services/hotel/
  setAvailableRooms?inHotelDate=010107&availableRooms=5

  This command will create a date where five rooms are available at the hotel. The date format is *ddmmyy* where *dd* means day, *mm* means month and *yy* means year.

- **Car Service:** The *Car Service* is configured by typing the following url into a Web browser:

  http://localhost:8080/axis2/services/car/
  setAvailableCars?inCarDate=010107&availableCars=3

  This command will create a date where three cars are available at a car rental company. The date format is *ddmmyy* where *dd* means day, *mm* means month and *yy* means year.

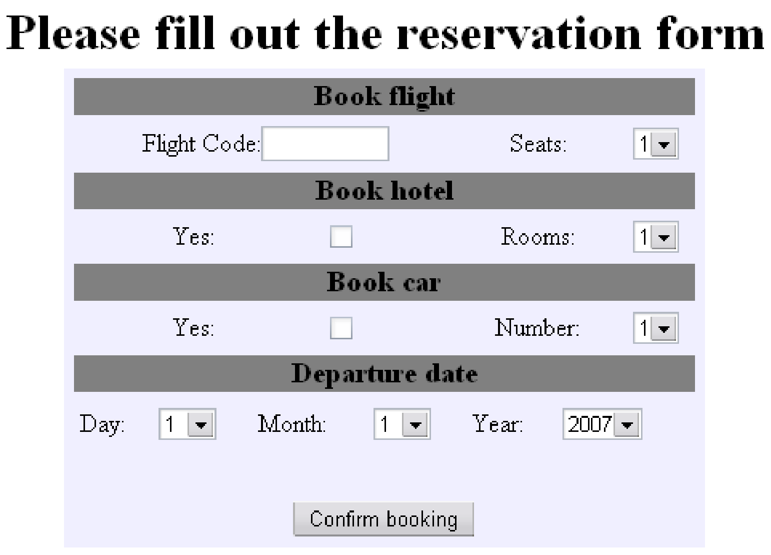
6.2 Running a real test

A test can be run once the service providers have been configured. The easiest way to run the test is to type the following url in a Web browser:

http://localhost:8080/travel/servlet/travelagency

This url should bring up a Web page in the Web browser that contains a reservation form which looks like the one in figure 6.2.

From here it is possible to choose between the different options in the reservation form.

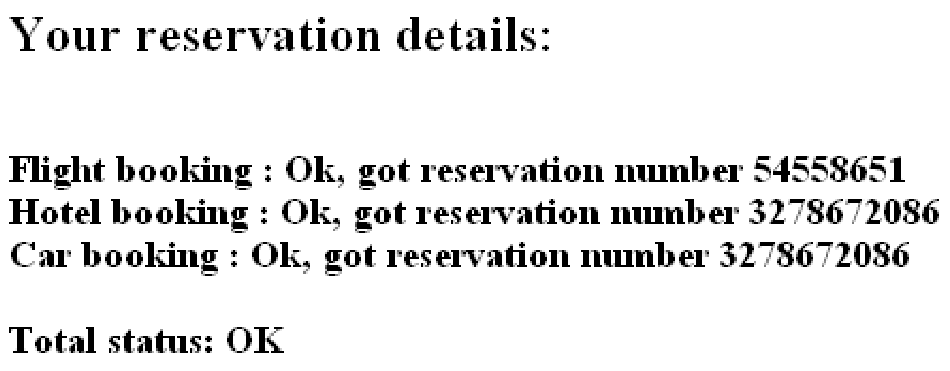

Please fill out the reservation form

Book flight			
Flight Code:		Seats:	1
Book hotel			
Yes:	☐	Rooms:	1
Book car			
Yes:	☐	Number:	1
Departure date			
Day: 1	Month: 1	Year: 2007	

Figure 6.2: A reservation form

6.2.1 A successful reservation

One way to get a successful reservation is to type *AA0004* into the textfield labeled *Flight Code*, and checking all the check boxes. Now, clicking on the button labeled *Confirm Booking* will cause an extended workflow specification to be created which looks like the one in Appendix C. This workflow specification can be executed directly by the workflow enactment service. When the workflow execution is finished, another Web page will be displayed in the Web browser that contains the results from the reservations. The results will look like figure 6.3, except that the reservation numbers may be different.


Your reservation details:

Flight booking : Ok, got reservation number 54558651
Hotel booking : Ok, got reservation number 3278672086
Car booking : Ok, got reservation number 3278672086

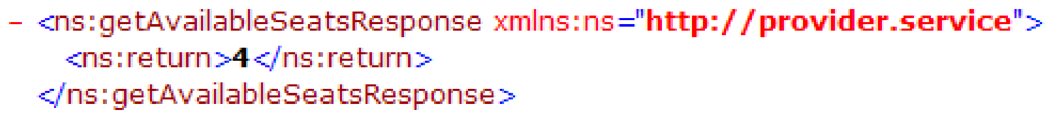
Total status: OK

Figure 6.3: A successful reservation

It is possible to see if the reservation of a flight seat really took place by typing the following url into the Web browser:

http://localhost:8080/axis2/services/airline/getAvailableSeats?inFlightCode=AA0004

If the Web browser is capable of displaying XML, then the result from this command should look similar to figure 6.4.



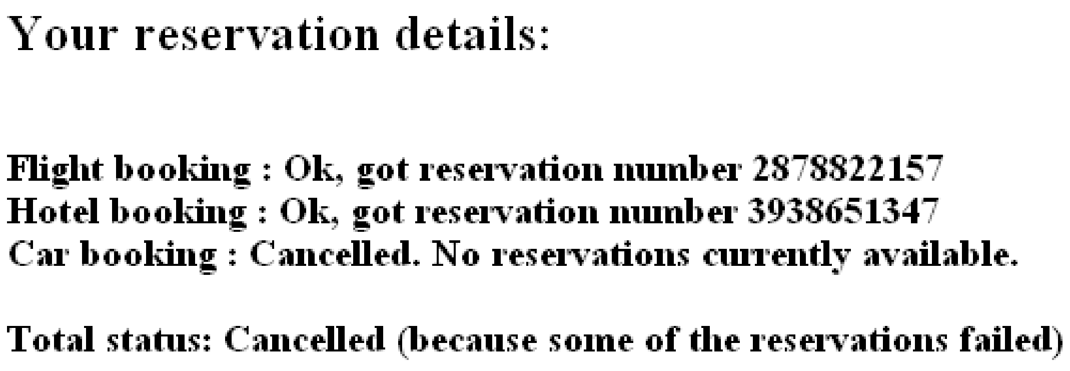
```
- <ns:getAvailableSeatsResponse xmlns:ns="http://provider.service">
    <ns:return>4</ns:return>
  </ns:getAvailableSeatsResponse>
```

Figure 6.4: Available flight seats

Figure 6.4 shows that there are four available flight seats left on the flight called *AA0004*.

6.2.2 A failed reservation

By continuing from the example with a successful reservation, and by going back to the Web page that contains the reservation form, then it is possible to choose a set of options which will cause the reservation to fail. This is possible by increasing the number of cars to above the available limit. Clicking the button labeled *Confirm Booking* will then cause a flight seat and a hotel room to be reserved, but these reservations will be compensated because of the failed car reservation. The results will look like figure 6.5, except that the reservation numbers may be different.



Your reservation details:

Flight booking : Ok, got reservation number 2878822157
Hotel booking : Ok, got reservation number 3938651347
Car booking : Cancelled. No reservations currently available.

Total status: Cancelled (because some of the reservations failed)

Figure 6.5: A failed reservation

Although the Web page displays the flight and hotel reservations as successfully completed, the overall result suggests that they did not succeed.

A failed reservation can also happen if one of the services is unreachable. A reason for this may be that the service has been removed from the server, but it can also be the case that the service is down.

6.3 Evaluation

The prototype behaves as expected for both tests which include a sucessful reservation and a failed reservation. The prototype is not adaptable in situations where one agent becomes unreachable. If the unreachable agent is used during a workflow execution, then the workflow execution will fail to respond with any results. However, if a service provider becomes unreachable, then any successfully completed activities will automatically be compensated. No tests for scalability and performance where performed, because these properties were not part of the goal for this thesis.

In addition, similar tests have also been carried out on a Local Area Network (LAN) where the service providers were located on a remote host. The results from these tests were identical to the other results.

Chapter 7

Conclusion

A workflow enactment service is the core of a WfMS. Therefore, having a workflow enactment service that is scalable, adaptable and robust is very important in order to provide reliable executions of distributed workflows. These properties can be achieved if the workflow enactment service uses a continuation-passing mechanism for distributed workflow enactment.

A Java prototype has been developed that uses a small scale application scenario to demonstrate continuation-passing orchestration of composite Web Services, using only certain types of flows which have been defined in a schema as shown in appendix A. Therefore, the goal for this thesis has been fulfilled. The continuation-passing mechanism, which the prototype uses, has been inspired by [24].

The prototype does not pre-allocate any resources prior to the workflow execution, which is one of the most important signatures of a continuation-passing enactment style. This allows multiple concurrent workflows to be executed by the workflow enactment service without wasting valuable resources.

Using a small scale application scenario is an important step towards having a full scale workflow enactment system which is based on the continuation-passing mechanism. In addition, the prototype enables recovery of failed workflow executions by using compensation activities to invert the effects of successfully completed activities.

However, some improvements can be made to the prototype. For example, the prototype can become more robust by avoiding failures caused by unreachable agents. If an agent has a list of other available agents, then this agent can randomly select another agent if one agent becomes unreachable.

A unique workflow reference number is created by the Web application called travel agency. This workflow reference number is assigned to a workflow specification before it is sent to one of the software agents. Problems can occur if there are multiple travel agencies which use the software agents simultaneously, because there can be duplicate workflow reference numbers assigned to different workflows. This situation is clearly not desirable, which means that there should be one standard interface to the workflow enactment service. As a result, one agent will be responsible for receiving workflow specifications from the travel agencies. This agent will assign a unique workflow reference number to each workflow. This is an improvement which enables multiple travel agencies to use the same workflow enactment service.

There are some known performance issues which relate to the Web service technology. For example, sending SOAP via HTTP improves the availability of Web services, because

HTTP easily passes through firewalls. However, HTTP is less efficient than raw sockets and other mechanisms [2]. Another performance issue is that SOAP uses XML, where XML requires parsing and interpreting which can result in processing bottlenecks. In addition, XML carries a lot of overhead in order to represent data [2]. This means that the size of SOAP messages becomes large, which can result in communication bottlenecks. Therefore, the flexibility of using Web services can have a large impact on performance. This becomes more obvious for a composite Web service which invokes other Web services.

The XML overhead does not benefit the performance of the prototype, because continuation-passing enactment requires that the rest of an execution should be sent during a remote transition. The rest of an execution can be a large portion of the workflow, which can make the messages which are sent between agents very large. Problems can occur when these messages are sent over the Internet, which is a WAN, because the available bandwidth can be limited. In a traditional decentralized workflow enactment service, the workflow engines will receive a separate part of the workflow. The worst case scenario for a continuation-passing enactment style is that every agent executes one single step of the workflow before sending the continuation onwards, but this is the price to pay in order to have an adaptable workflow enactment service that uses a continuation-passing enactment style for distributed workflows.

However, there are attempts to encode an XML document into a binary data format [23], which is much faster to parse than plain text. This can improve the performance of a continuation-passing enactment of distributed workflows.

The lessons learned during the prototype development stages have cast a new light on the importance of having open source projects like Apache Axis2, Apache Tomcat, Apache Ant, Apache Velocity and Eclipse SDK. These projects have made it possible to realize the Java prototype which demonstrates a continuation-passing enactment of distributed recoverable workflows. However, it was a challenge to debug the prototype when something went wrong, because there could have been a bug present in the workflow specification long before problems began to surface. Having a good intuition for how the workflow was going to be executed by the workflow enactment service was an important part of the workflow design phase. Therefore, the only way to debug a workflow specification, when something went wrong, was through the use of human intuition.

References

[1] Ryan A. What is a business process? http://www.visualocity.net/visualocity/articles/whatIsABusinessProcess.jsp, September 2006.

[2] Britton C. and Bye P. *IT Architecture and Middleware, Second Edition*. Addison-Wesley, 2004.

[3] Chappell D. and Jewell T. *Java Web Services*. O'Reilly, March 2002.

[4] Hollingsworth D. The workflow reference model. http://www.wfmc.org/standards/docs/tc003v11.pdf, January 1995.

[5] Sosnoski D. Digging into axis2: Axiom. http://www.ibm.com/developerworks/webservices/library/ws-java2/, 2006.

[6] Piedad F. and Hawkins M. *High Availability: Design, Techniques, and Processes*. Prentice Hall, Desember 2000.

[7] Apache Software Foundation. Apache tcpmon. http://ws.apache.org/commons/tcpmon/index.html.

[8] Apache Software Foundation. The apache ant project. http://ant.apache.org/, 2007.

[9] Apache Software Foundation. The apache axis2/java project. http://ws.apache.org/axis2/index.html, 2007.

[10] Apache Software Foundation. The apache tomcat project. http://tomcat.apache.org/, 2007.

[11] Apache Software Foundation. The apache velocity project. http://velocity.apache.org/engine/releases/velocity-1.5/, 2007.

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

[13] W3C DOM Interest Group. Document object model (dom). http://www.w3.org/DOM/, 2005.

[14] Lunn K. *Software Development with UML*. Palgrave Macmillan, 2003.

[15] Sun Developer Network. Java archive. http://en.wikipedia.org/wiki/JAR_(file-format), 2007.

[16] Sun Developer Network. Java technologies. http://java.sun.com/, 2007.

[17] OASIS. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, April 2007.

[18] Harold E. R. and Means W. S. *XML in a Nutshell, 3rd Edition*. O'Reilly, September 2004.

[19] Weerawarana S., Curbera F., Leymann F., Storey T., and Ferguson D. *Web Services Platform Architecture*. Prentice Hall, 2006.

[20] Open Source. Eclipse classic. http://www.eclipse.org/, 2007.

[21] Mallalieu T. and Carriere J. Enterprise interoperability: .net and j2ee. http://msdn2.microsoft.com/en-us/library/ms954598.aspx, 2004.

[22] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT press, 2004.

[23] Wikipedia. Binary xml. http://en.wikipedia.org/wiki/Binary_XML.

[24] Yu and Yang. Continuation-passing enactment of distributed recoverable workflows. In proceedings of ACM SAC'07.

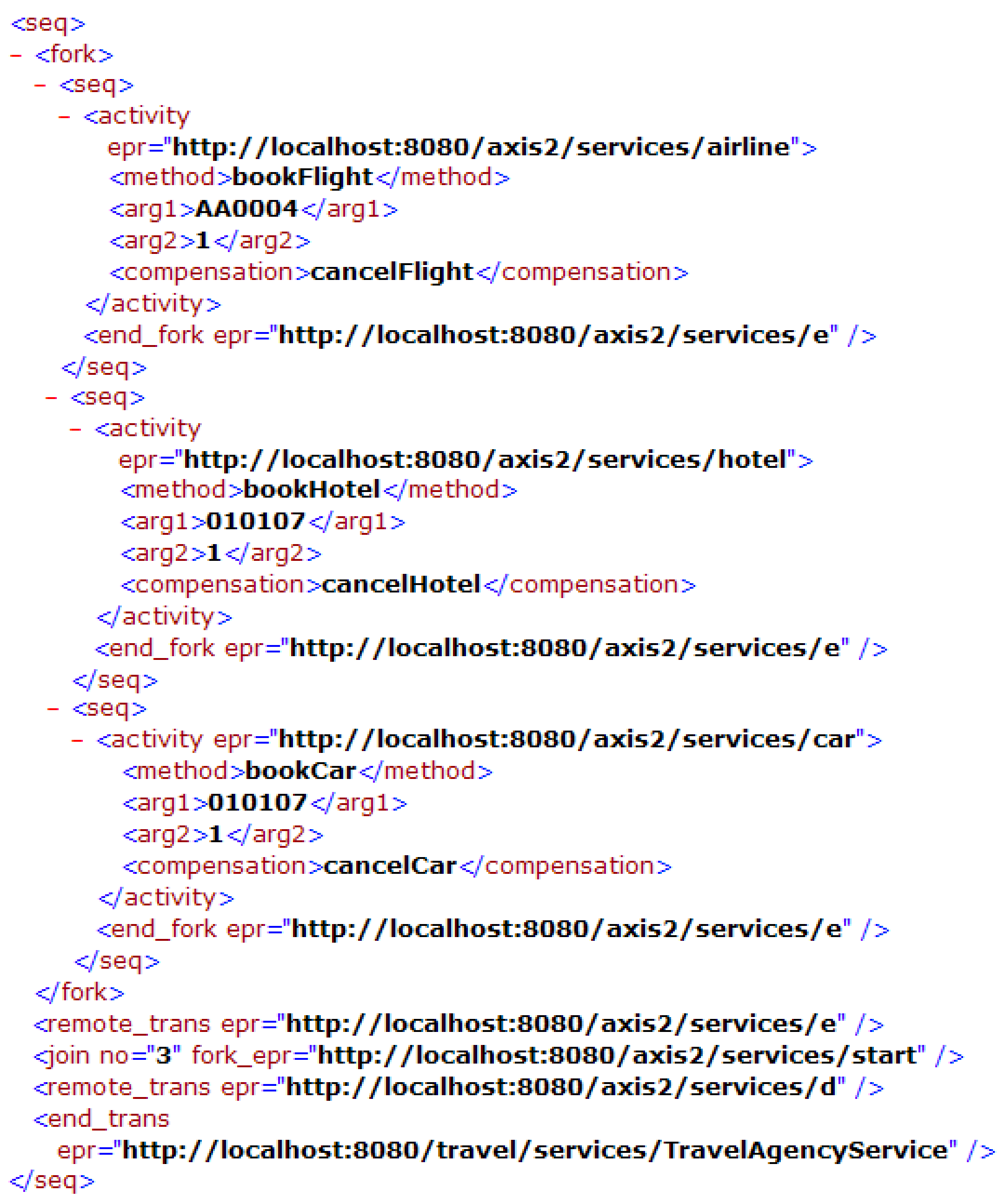
Appendix A

XML schema for workflow specification

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://provider.service/cekk"
    targetNamespace="http://provider.service/cekk"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

    <xs:complexType name="cekkType">
        <xs:sequence>
            <xs:element name="c" type="tns:cType" />
            <xs:element name="e" type="tns:eType" />
            <xs:element name="ks" type="tns:ksType" />
            <xs:element name="kf" type="tns:kfType" />
        </xs:sequence>
        <xs:attribute name="id" type="xs:integer" use="required" />
        <xs:attribute name="fork_epr" type="xs:string" use="required" />
    </xs:complexType>

      <xs:complexType name="cType">
        <xs:sequence>
            <xs:element name="seq" type="tns:seqType" />
        </xs:sequence>
    </xs:complexType>

      <xs:complexType name="eType">
        <xs:sequence>
            <xs:element name="seq" type="tns:seqType" />
        </xs:sequence>
    </xs:complexType>

     <xs:complexType name="ksType">
        <xs:sequence>
            <xs:element name="seq" type="tns:seqType" />
        </xs:sequence>
    </xs:complexType>

     <xs:complexType name="kfType">
        <xs:sequence>
            <xs:element name="seq" type="tns:seqType" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="seqType">
        <xs:sequence>
            <xs:element name="activity" type="tns:activityType" />
            <xs:element name="fork" type="tns:forkType" />
            <xs:element name="end_fork" type="tns:end_forkType" />
            <xs:element name="join" type="tns:joinType" />
            <xs:element name="remote_trans" type="tns:remote_transType" />
            <xs:element name="end_trans" type="tns:end_transType" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="activityType">
        <xs:sequence>
            <xs:element name="method" type="xs:string" />
            <xs:element name="arg1" type="xs:string" />
            <xs:element name="arg2" type="xs:string" />
            <xs:element name="compensation" type="xs:string" />
        </xs:sequence>
        <xs:attribute name="epr" type="xs:string" use="required" />
    </xs:complexType>

    <xs:complexType name="remote_transType">
        <xs:sequence>
            <xs:element name="cekk" type="tns:cekkType" />
        </xs:sequence>
        <xs:attribute name="epr" type="xs:string" use="required" />
    </xs:complexType>

    <xs:complexType name="end_transType">
        <xs:sequence>
            <xs:element name="cekk" type="tns:cekkType" />
        </xs:sequence>
        <xs:attribute name="epr" type="xs:string" use="required" />
    </xs:complexType>

    <xs:complexType name="end_forkType">
        <xs:sequence>
            <xs:element name="cekk" type="tns:cekkType" />
        </xs:sequence>
        <xs:attribute name="epr" type="xs:string" use="required" />
    </xs:complexType>

    <xs:complexType name="forkType">
        <xs:sequence>
            <xs:element name="cekk" type="tns:cekkType" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="joinType">
        <xs:attribute name="no" type="xs:integer" use="required" />
        <xs:attribute name="fork_epr" type="xs:string" use="required" />
    </xs:complexType>

    <xs:element name="cekk" type="tns:cekkType" />
</xs:schema>
```

Appendix B

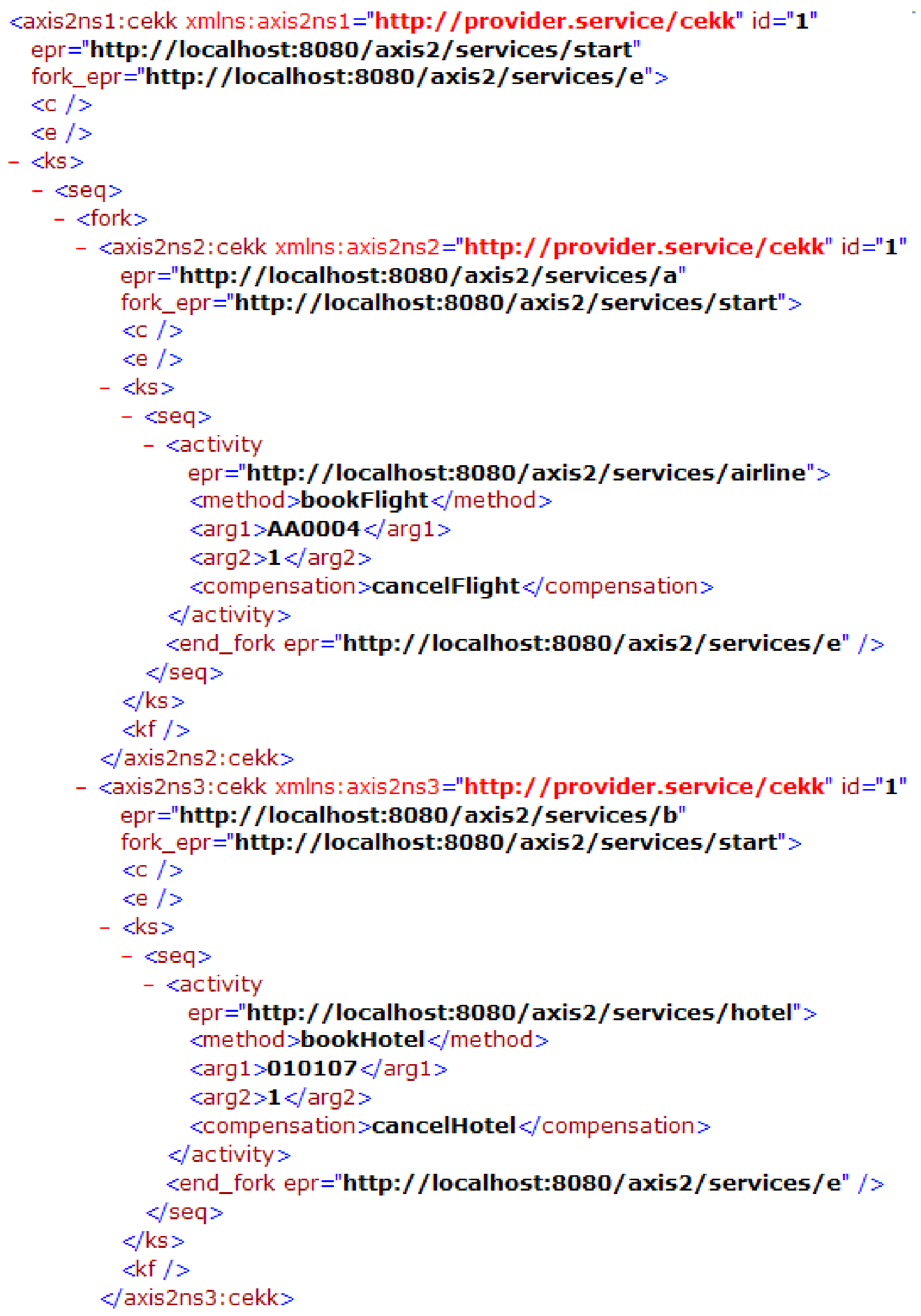
A sample workflow specification



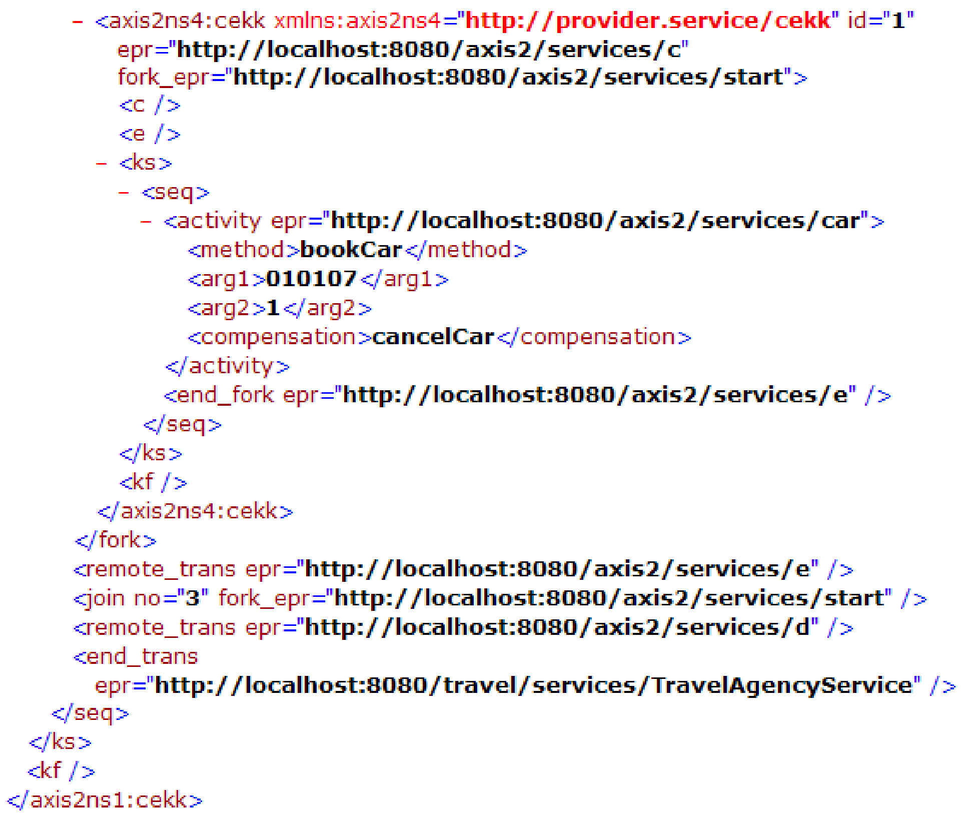
```
<seq>
- <fork>
  - <seq>
    - <activity
        epr="http://localhost:8080/axis2/services/airline">
        <method>bookFlight</method>
        <arg1>AA0004</arg1>
        <arg2>1</arg2>
        <compensation>cancelFlight</compensation>
      </activity>
      <end_fork epr="http://localhost:8080/axis2/services/e" />
    </seq>
  - <seq>
    - <activity
        epr="http://localhost:8080/axis2/services/hotel">
        <method>bookHotel</method>
        <arg1>010107</arg1>
        <arg2>1</arg2>
        <compensation>cancelHotel</compensation>
      </activity>
      <end_fork epr="http://localhost:8080/axis2/services/e" />
    </seq>
  - <seq>
    - <activity epr="http://localhost:8080/axis2/services/car">
        <method>bookCar</method>
        <arg1>010107</arg1>
        <arg2>1</arg2>
        <compensation>cancelCar</compensation>
      </activity>
      <end_fork epr="http://localhost:8080/axis2/services/e" />
    </seq>
  </fork>
  <remote_trans epr="http://localhost:8080/axis2/services/e" />
  <join no="3" fork_epr="http://localhost:8080/axis2/services/start" />
  <remote_trans epr="http://localhost:8080/axis2/services/d" />
  <end_trans
    epr="http://localhost:8080/travel/services/TravelAgencyService" />
</seq>
```

Appendix C

An extended version of the sample workflow specification



```
<axis2ns1:cekk xmlns:axis2ns1="http://provider.service/cekk" id="1"
  epr="http://localhost:8080/axis2/services/start"
  fork_epr="http://localhost:8080/axis2/services/e">
  <c />
  <e />
- <ks>
  - <seq>
    - <fork>
      - <axis2ns2:cekk xmlns:axis2ns2="http://provider.service/cekk" id="1"
          epr="http://localhost:8080/axis2/services/a"
          fork_epr="http://localhost:8080/axis2/services/start">
          <c />
          <e />
        - <ks>
          - <seq>
            - <activity
                epr="http://localhost:8080/axis2/services/airline">
                <method>bookFlight</method>
                <arg1>AA0004</arg1>
                <arg2>1</arg2>
                <compensation>cancelFlight</compensation>
              </activity>
              <end_fork epr="http://localhost:8080/axis2/services/e" />
            </seq>
          </ks>
          <kf />
        </axis2ns2:cekk>
      - <axis2ns3:cekk xmlns:axis2ns3="http://provider.service/cekk" id="1"
          epr="http://localhost:8080/axis2/services/b"
          fork_epr="http://localhost:8080/axis2/services/start">
          <c />
          <e />
        - <ks>
          - <seq>
            - <activity
                epr="http://localhost:8080/axis2/services/hotel">
                <method>bookHotel</method>
                <arg1>010107</arg1>
                <arg2>1</arg2>
                <compensation>cancelHotel</compensation>
              </activity>
              <end_fork epr="http://localhost:8080/axis2/services/e" />
            </seq>
          </ks>
          <kf />
        </axis2ns3:cekk>
```



```
      - <axis2ns4:cekk xmlns:axis2ns4="http://provider.service/cekk" id="1"
          epr="http://localhost:8080/axis2/services/c"
          fork_epr="http://localhost:8080/axis2/services/start">
          <c />
          <e />
        - <ks>
          - <seq>
            - <activity epr="http://localhost:8080/axis2/services/car">
                <method>bookCar</method>
                <arg1>010107</arg1>
                <arg2>1</arg2>
                <compensation>cancelCar</compensation>
              </activity>
              <end_fork epr="http://localhost:8080/axis2/services/e" />
            </seq>
          </ks>
          <kf />
        </axis2ns4:cekk>
      </fork>
      <remote_trans epr="http://localhost:8080/axis2/services/e" />
      <join no="3" fork_epr="http://localhost:8080/axis2/services/start" />
      <remote_trans epr="http://localhost:8080/axis2/services/d" />
      <end_trans
        epr="http://localhost:8080/travel/services/TravelAgencyService" />
    </seq>
  </ks>
  <kf />
</axis2ns1:cekk>
```

Appendix D

A list of flight service method invocations

Create a flight called *AA0004* with four available seats:

> http://localhost:8080/axis2/services/airline/
> setAvailableSeats?inFlightCode=AA0004&availableSeats=4

Get the number of available seats in flight *AA0004*:

> http://localhost:8080/axis2/services/airline/getAvailableSeats?inFlightCode=AA0004

Get the number of flights:

> http://localhost:8080/axis2/services/airline/getNumberOfFlights

See how many reservations there are:

> http://localhost:8080/axis2/services/airline/getNumberOfOrders

Check if there is a flight reservation with the reference number *2878822157*:

> http://localhost:8080/axis2/services/airline/getFlightOrders?inOrderID=2878822157