Dependable composition of Web services

Ferda Tartanoglu

N^{O} Ordre de la thèse :

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE PARIS 6

pour obtenir

le grade de : *DOCTEUR DE L'UNIVERSITÉ DE PARIS 6*

Mention : **Informatique**

PAR
Galip Ferda TARTANOĞLU

Équipe d'accueil : **INRIA, Projet ARLES**

École Doctorale : **Informatique, Télécommunications et Electronique de Paris**

TITRE DE LA THÈSE :

Composition Sûre de Fonctionnement de Services Web

SOUTENUE LE 9 / 12 / 2005 devant la commission d'Examen

COMPOSITION DU JURY

Nicole LÉVY (Université de Versailles Saint-Quentin-en-Yvelines)	Rapporteur
Alexander ROMANOVSKY (Université de Newcastle upon Tyne)	Rapporteur
Pierre SENS (Univerité Paris 6)	Examinateur
Apostolos ZARRAS (Université de Ioannina)	Examinateur
Valérie ISSARNY (INRIA)	Directeur de Thèse

Résumé

Les services Web offrent un certain nombre de propriétés intéressantes pour le développement des systèmes distribués ouverts, construits par la composition de services autonomes. Cependant, les systèmes résultants doivent offrir des propriétés non-fonctionnelles et en particulier des propriétés de sûreté de fonctionnement pour être adopté par les utilisateurs, et notamment pour leur utilisation effective dans le domaine de l'e-business. Cette thèse propose une méthode et des intergiciels associés pour la composition sûre de fonctionnement de services autonomes. Notre contribution porte sur une extension des interfaces des systèmes composés par leur propriétés de sûreté de fonctionnement, une définition d'un langage de composition spécifique pour assurer la sûreté de fonctionnement du service composite et sur un support d'exécution pour la mise en œuvre des mécanismes de tolérance aux fautes. L'extension proposée aux interfaces des services Web est spécifié par un langage de conversation qui permet de définir les règles pour appeler les opérations offertes par les services Web individuelles. Les propriétés relatives au recouvrement sont associées à ces conversations, permettant de raisonner sur la stratégie de recouvrement qui peut être mise en œuvre dans la composition de service. En effet, les comportement de recouvrement des services composites dépendent des propriétés de recouvrement des services composés, ces derniers devant effectuer les actions de recouvrement en présence de fautes. Cette thèse propose ensuite un langage de composition déclaratif qui permet de développer des services composites en termes d'actions atomiques. Nous permettons la spécification du traitement d'exceptions impliquant plusieurs services Web au niveau de la composition, permettant en particulier d'intégrer des services non-sûres dans la composition tout en garantissant la sûreté de fonctionnement du service composite. Nous montrons que la tolérance aux fautes peut être obtenue comme une propriété émergente de l'agrégation de plusieurs services, potentiellement non-sûres.

Abstract

Web services offer a number of valuable features towards supporting the development of open distributed systems, built out of the composition of autonomous services. Nonetheless, the resulting systems must offer a number of non-functional properties and in particular dependability-related ones, for acceptance by users, including effective exploitation in the e-business domain. This thesis proposes a method and associated middleware services for the dependable composition of autonomous systems. Our contribution subdivides into the extension of the interfaces of systems with their dependability capabilities, a definition of a specific composition language oriented towards providing dependability for the composite system and a runtime support that implements fault tolerance mechanisms. The proposed extension of Web service interfaces is specified using a conversation language that sets the rules for calling the operations offered by individual Web services. Recovery-related properties are associated to these conversations, enabling to reason about the recovery strategy that can be implemented in the service composition. Indeed, the recovery behaviour of composite services depend upon the recovery properties of the composed Web services, since the latter must ultimately perform some recovery actions in the presence of faults. This thesis further proposes a declarative composition language, which allows developing composite Web services in terms of dependable actions. We allow exception handling involving several Web services to be specified at the composition level, enabling in particular to integrate non-dependable Web services in the composition while still supporting dependability of the composite service. We show that fault tolerance can be obtained as an emergent property of the aggregation of different, potentially non-dependable, services.

Remerciements

Je tiens à exprimer toute ma reconnaissance à Madame Valérie Issarny pour son encadrement, ses nombreux conseils et son soutien constant tout au long de ma thèse.

Je tiens à exprimer ma profonde gratitude à Monsieur Pierre Sens, qui m'a fait l'honneur de présider le jury de thèse de doctorat, pour l'intérêt et le soutien chaleureux dont il a toujours fait preuve.

Je suis reconnaissant à Madame Nicole Levy et à Monsieur Alexander Romanovsky d'avoir acceptés d'être rapporteurs de ma thèse. Leurs commentaires et leurs questions m'ont permis de clarifier ma rédaction et m'ont donné de nouvelles pistes de réflexion.

Je remercie Monsieur Apostolos Zarras pour avoir accepté de faire partie de mon jury de thèse. Pour cela, ainsi que pour ses conseils avisés, notamment concernant la rédaction scientifique, je lui exprime ma profonde gratitude.

Je remercie tous les chercheurs, ingénieurs, thésards, stagiaires et membres du projet Arles de l'INRIA Rocquencourt pour leur amitié et leur aide pendant ces années de thèse.

Je tiens à témoigner tout particulièrement ma sympathie et ma reconnaissance à Françoise avec qui j'ai partagé le bureau pendant ces années, et avec qui j'ai eu tant de discussions fructueuses.

Enfin, pour leur soutien sans faille et permanent, je tiens à remercier de tout coeur mes parents et Bahar auxquels je dédie mon mémoire de thèse.

Table des matières

Table des figures

Première partie

Dependable Composition of Web Services

I Introduction

Service-oriented computing aims at the development of highly-autonomous, loosely-coupled systems that are able to communicate, compose and evolve in a dynamic and heterogeneous environment. Autonomous systems that are developed and administered by distinct entities hide their implementation details with well-defined interfaces that are made publicly available. Interfaces are written in a standardized form enforcing interoperability across diversely implemented systems. Applications that are deployed over the Internet by an increasing number of organizations are typical examples of such autonomous and loosely coupled systems. These applications should be able to inter-operate without loose of their autonomy and should be able to adapt to the changing environment where devices and resources move, components appear, disappear and evolve. They should also deal with the increasing requirements of service consumers on quality of service guarantees. All these requirements raise a number of challenges, motivating the definition of new architectural principles. One such challenge is to provide dependability of the composition of autonomous and potentially non-dependable systems. We propose in this thesis, a method and associated middleware services for the dependable composition of autonomous systems. Our contribution subdivides into the extension of the interfaces of systems with their dependability capabilities, a definition of a specific composition language oriented towards providing dependability for the composite system and a runtime support that implements fault tolerance mechanisms.

I.1 Service-Oriented Architectures

Various software architectures and technologies have been proposed over the last 30 years for easing the development and deployment of distributed systems (e.g., middleware for distributed objects [Emmerich, 2000]). However, the generalization of the Internet and the diversification of networked devices have led to

the definition of a new computing paradigm: the Service-Oriented Architecture (SOA), which allows developing software as a service delivered and consumed on demand [Elfatatry and Layzell, 2004, Papazoglou and Georgakopoulos, 2003]. The benefit of this approach lies in the looser coupling of the software components making up an application, hence the increased ability to making evolve systems as application-level requirements and the networked environment change.

I.1.1 Services

A *service* is defined as a unit of work performed by a self-contained software system and delivered to another software system. The software system that provides the service is called the *service provider*, and the consumer of the service is called the *service requester*. In general, service providers, are deployed independently of service requesters, and make available the services they want to provide over a public or private network.

To achieve interoperability among systems, service providers expose a platform-independent *service contract* that describes *what* is the service, *how* a service requester should interact with the provider to get the service and the Service Level Agreement (SLA) that comprises, in particular, quality-of-service (QoS) attributes (performance, security, transactional behaviour, etc.). Using service contracts, service requesters discover and select service providers that can deliver a service satisfying their functional and non-functional requirements. Communication between service requesters and providers is mostly done by message exchange, which enforces loose coupling as it enables asynchronous communication among parties. We distinguish *request messages* sent by service requesters and *response messages* sent by service providers. Other message types include protocol messages such as acknowledgments for realizing reliable messaging, and fault messages such as exceptions sent as responses to malformed request messages. An interaction is then defined by the sum of all messages sent and received between the requester of the service and its provider for delivering the service. The most basic interaction is the one-way message sent or received by a service. RPC-like messaging can be realized by combining a request message with a response message and more complex interactions can be defined by grouping and ordering several one-way messages. Like the *service contract* that is described in a platform-independent language, messages are also encoded in a platform- and language-neutral structure for interoperability.

In the service-oriented architecture, system components evolve continuously and independently of each other. New services appear, existing services disappear

permanently or get unavailable temporarily, services change their interfaces, etc. Moreover, service requesters' functional and non-functional requirements may change over time depending on the context. Adaptation to these changes is thus a key feature of a service-oriented architecture, which is supported thanks to *service discovery* and *dynamic binding*. Service providers make available their offered services by publishing them using a service discovery protocol, e.g., by registering services in a service registry. Service requesters use the service discovery protocol for locating services that match their requirements. Adaptation of service requesters to the changing environment is enforced if the selection and localization of services are made during execution, through dynamic binding with matching services, which allows requesters to choose a service provider at runtime. Note that the discovery service may be centralized as well as distributed (e.g., supported by all the service hosts), and may further adhere to either a passive (led by service provider) or active (led by service requester) discovery model [Bromberg and Issarny, 2005].

Summarizing, a *service-oriented architecture* is defined as a collection of service requesters and providers, interacting with each other according to agreed contracts. Service requesters usually locate service providers at execution time using some service discovery protocol. A typical interaction in a service-oriented architecture involving a service requester, service providers, and a centralized service registry is depicted in Figure I.1 where boxes represent system components, unidirectional arrows represent one-way messages and bidirectional arrows represent interactions formed by multiple messages:

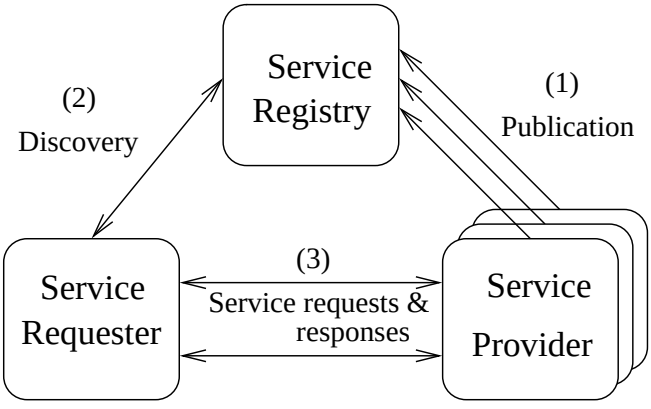


Figure I.1: Interactions

(1) Service providers register the description of the services they offer. The description of the service includes the provided service contract and binding information such as the network endpoint address and the interaction

protocol, which describes the messaging behaviour of the service, to be taken into account for correctly interacting with the service.

(2) A service requester queries at runtime the service registry for locating a provider that can deliver the service it needs. The service registry then returns to the requester descriptions of available services –if any– whose provided contracts match the requirements of the service requester.

(3) The service requester selects a service provider among those discovered and initiates interactions by sending it a request message. The interaction between the requester and the provider then continues according the terms of the agreed service contract.

It is important to note that the interactions between the service requester and provider may follow an interaction protocol involving enhanced middleware-related services such as replication, security, and transaction management. The service interfaces should include the interaction protocols and associated non-functional properties, which need to be understood and adhered by all the interacting parties. In particular, for reasoning about dependability, error recovery capabilities of services such as their transactional behaviour, should make part of the service interface.

The Web service architecture[1] has become a convenient enabling technology for building service-oriented architectures due to huge efforts on the standardization of its elements and developments of supporting platforms. According to the definition of the W3C Consortium, a Web service is a software system identified by a URI[IETF, 1998], whose public interfaces are described using XML-based languages and which interacts with other systems in a manner prescribed by the service interface, using XML-based messages conveyed by standard Internet transport protocols like HTTP. More precisely, the W3C has defined the Web Service Description Language (WSDL) for describing interfaces[W3C, 2005], the Simple Object Access Protocol (SOAP) for defining the format and the processing rules of messages[W3C, 2003b] and the Oasis Consortium's Universal Description Discovery and Integration (UDDI) specification defines an API for publishing and discovering Web services in centralized registries [OASIS, 2004c].

[1] World Wide Web Consortium. Web Services Architecture Working Group. `http://www.w3.org/TR/ws-arch/`

I.1.2 Service composition

Services that are described through well-defined interfaces can be used to build new composite services, irrespective of technical details regarding the underlying platform and the implementation of component services, which are called *composed services* throughout this document. A service built using service composition is called a *composite service*, and can in its turn, be part of a larger composition. The composite service can deliver new functionalities with new properties thanks to the composition.

As an illustration for the composition of autonomous services, we take the travel agency case study, which is often used in illustrating integration of autonomous systems. The travel agency is a composite service that assists the user in booking complete trips by accessing existing travel services. There are several challenges that a designer of such a composite application faces. Interactions with concurrently running autonomous services should be correctly coordinated to get the expected result from each service and provide the advertised integrated service. The resulting composite service should guarantee a high level of dependability despite the use of autonomous and potentially undependable composed services. In particular, the travel agent composite service has to deal with problems occurring during execution such as partially completed trip reservations, transparently to the service requester.

Several properties of the Web services architecture must be taken into account while addressing the above issues. Web services are decentralized in architecture and in administration. Therefore, individual Web services can have different characteristics (e.g., transactional behaviour, concurrency policies, access rights), which may not be compliant with each other. Moreover, Web services communicate using Internet transport protocols (e.g. HTTP, SMTP) and interacting with them requires dealing with limitations of the Internet such as access latency, timeouts and lost requests as well as with security issues.

Although the modularity and interoperability of the Web services architecture enable complex distributed services to be easily built by assembling several component services into one composite service, there clearly is a number of research challenges in supporting the thorough development of dependable composite Web services. This calls for developing new architectural principles of building dependable composite services, in general, and for studying specialized connectors "gluing" Web services, in particular, so that the resulting composition can deal with failures occurring at the level of the individual component services.

I.2 Dependable service composition

Building dependable composite services requires first to understand the dependability properties of individual services. A dependable composite service can then be built according to the properties of the composed services and of the dependability requirements of the composite service. Finally, there is a need for a runtime support that can implement the adequate dependability mechanisms.

I.2.1 Dependability properties of individual services

With service composition, offered dependability mechanisms of individual Web services can be combined to meet the dependability requirements of composite applications. Indeed, the recovery behaviour of composite services depends upon the recovery properties of the composed services, since the latter must ultimately perform some recovery actions (e.g., compensation) in the presence of faults, which require adequate specification of the individual Web services. Expressing the behaviour of Web services in the presence of faults is partly addressed in the definition of service interfaces through fault messages. However, this does not specify complex recovery properties such as compensating operations, as exploited by advanced transaction models for Web services. Other attempts describe the recovery behaviour of Web service operations by annotating service operations with pre-defined meta-data. However, these approaches are not sufficient for comprehensively expressing the recovery behaviour of a service. The error recovery mechanism that is implemented by the service requester-side (i.e., the composite service) often involves calling multiple operations on the service provider-side. Moreover, for delivering the target recovery property, the service provider may require that the operations be invoked in a specific order and under some conditions. This then suggests specifying the recovery properties of Web services at the level of the external visible behaviour of Web services using an adequate description language.

I.2.2 Dependability of composite services

The developer of composite Web services has as input the interfaces of different Web services. The functional interface of a Web service gives the names of offered operations and related messages. Additionally, non-functional properties (performance, security, transactional capabilities, access policy etc.) related

to the provided service can be expressed in complementary standardized documents. The developer has then in charge to build a composite service based on the information that is provided at the interfaces. However, the interfaces of composed Web services may be incomplete or non-compatible with each other, making it difficult to reason about the global properties of a composite application. Some properties, e.g., non-functional properties, exposed at the interface of each Web service can further loose their meaning when composed. Moreover, Web services can return responses at runtime that do not exactly conform with what is advertised at the interfaces, transgressing the service contract. The development of dependable composite Web services is further complicated by the fact that composed Web services expose different error recovery behaviours or not at all, which raises challenging issues in specifying composition processes and in particular behaviour of composite services in the presence of faults.

The choice of the fault tolerance mechanisms that should be used in the development of composite services depends on applications, and very often there is a need to combine different error recovery mechanisms that should be expressed at the composition level. Although there exist several Web services composition languages that include constructs for specifying fault tolerance requirements of the composite Web service, they mainly target backward error recovery by the exploitation of advanced transactions or the introduction of specific transaction protocols [Tartanoglu et al., 2003a]. These solutions assume that composed Web services that are integrated are compliant with dependability requirements of the composite Web services. On the other hand, forward error recovery is exploited for internal exception handling or for specifying compensating actions. However, applying backward error recovery is not always possible in the context of Web services. Furthermore, applying forward error recovery in a composition often needs involving several Web services in the recovery process, which needs to be coordinated.

A Web service composition language, which is used to describe the execution workflow of a composite service, should allow designing dependable composite services by allowing to declare the dependability behaviour of the composite Web service and the dependability requirements expected from composed Web services. Dependability requirements can then be reached by composing several Web services, with different dependability properties, using the most adequate fault tolerance mechanism. A service composition language is needed to declare the behaviour of the composite service when error occurs, by composing different error recovery behaviours of composed Web services. Backward and forward error recovery should both be exploitable for an efficient usage of all Web services capabilities.

I.2.3 Fault tolerance mechanisms

In general, the choice of fault tolerance techniques to be exploited for the development of dependable systems depends very much on the fault assumptions and on the system's characteristics and requirements. Several types of faults can occur during the interaction with Web services, influencing the reliability of the whole system. These faults include but are not limited to: (i) faults occurring at the level of the Web services, which may be notified by error messages, (ii) faults occurring at the underlying platform (e.g., hardware faults, communication errors, timeouts), and (iii) faults due to on-line upgrades of component services and/or of their interfaces. In addition, unavailability is a major issue in addressing dependability in Web service applications. Web services may be unavailable for an unknown reason and for an unknown amount of time. Moreover, the overall network status and server loads may cause extensively long delays on responses from Web service servers.

The runtime support on top of which executes the composite service is responsible to detect faults and notify the fault at the application level. Fault messages received from Web services can be directly reported to the application, while transport-level faults can be mapped to internal exceptions for the composite service. Another role of the runtime support is to implement the fault tolerance mechanisms according to the recovery behaviour of the composite service and considering the recovery support of individual Web services. In particular, adapting the fault tolerance mechanisms transparently to the composite service is crucial since composed Web services instances available in a given environment and their recovery properties are not always known at design-time. Adaptation would in addition increase the number of Web services that can be integrated in the composition. To achieve this, we need a runtime support able to dynamically discover Web services and their recovery properties and execute the adequate mechanisms to implement the expected recovery behaviour of the composite application.

I.3 Contributions

The objective of this work is to build dependable composite Web services out of potentially non-dependable, component Web services that may fail or behave not as expected for certain reasons inherent to the unpredictable nature of the Internet. To achieve this goal, three issues were identified, as surveyed in the previous section. First, building dependable composite services requires reasoning about

the dependability properties of individual composed Web services. Second, the standard and exceptional behaviour of the composite Web service should be defined at the composition level, using an adequate composition language. Finally, different fault tolerance mechanisms should be combined at the implementation level to benefit from the recovery supports of the composed Web services. The contributions of this work to the dependability in the Web services architecture lie in addressing the above issue:

- A new service description language for specifying the recovery behaviour of individual Web services [Tartanoglu and Issarny, 2005].
- A Web service composition language and an associated dedicated recovery model [Tartanoglu et al., 2003c].
- A runtime support that executes composite Web services developed using our language and that implements fault tolerance mechanisms for the recovery model.

A first contribution of this thesis is the definition of a declarative language that enables the specification of the individual behaviour and capabilities of Web services that are relevant to providing dependability. We introduce the WS-RESC language in order to support reasoning about which recovery strategies can be implemented at the composition level. Using WS-RESC, the interface of each Web service can be complemented with additional recovery-related informations such as the the correct ordering of interactions that is assumed by the service, the transactional behaviour, the support for concurrency, etc. This specification is then used by the composite Web service during the service discovery phase to select Web services that can implement a specific fault tolerance mechanisms and to verify during the execution phase the correct usage of the service by the composite Web service. Furthermore, the fault tolerance mechanism that is implemented can be customized according to the capabilities of the composed Web services. In particular, concurrency control used for controlling accesses to composed Web services by the composite Web service is adapted according to the WS-RESC description to increase concurrency.

This thesis further proposes the composition language WSCAL for the specification of dependable composite Web services. The language offers constructs for specifying the recovery behaviour of the composite Web service for both backward and forward error recovery. The approach is inspired by the Coordinated Atomic Action model [Xu et al., 1995], which is adapted to the context of Web

services for providing a base structuring mechanism for developing fault tolerant composite Web services. The composition language is used to describe the composition process, which is used to generate an executable code implementing a composite Web service out of composed Web services that are dynamically discovered. The language enables specifying the concurrency support and the exceptional behaviour of the composite Web service. Error recovery is realized according to a coordinated exception handling model where several of the composed Web services can be involved in the recovery of a single exception. Concurrently raised exceptions are taken into account by choosing appropriate exception handlers following a concurrent exception resolution scheme. The language and its associated execution model, enables building dependable composite Web services by providing dependability at the composite application level while allowing benefiting from recovery capabilities of individual services.

Finally, a runtime support is implemented. The runtime support implements recovery mechanisms for realizing the recovery behaviour expressed in the specification of the composite Web service according to the capabilities of composed Web services. Coordinated exception handling is used to implement the specific recovery strategies including both backward and forward error recovery mechanisms, intended to deliver the expected guarantees specified by the composite Web service developer. The runtime support further includes middleware services for: (i) controlling concurrent accesses to Web services according to the isolation requirements of the composite service expressed in WSCAL and to the properties of the composed Web services expressed in WS-RESC, and (ii) controlling the correct invocation sequences assumed by Web services with on the fly verification.

I.4 Document structure

This dissertation is organized as follows. Chapter II presents the background work underlying our solution to the dependable Web services composition. More specifically, it presents the different types of service-oriented languages for specifying composite services and related work in providing dependability. Chapter III introduces a service-oriented language for specifying recovery capabilities of individual Web services. The introduced language is intended to be used for dynamically selecting matching Web services, verifying correctness of the composition at runtime and for customizing recovery behaviour at the middleware layer. In chapter IV, a composition language for building dependable composite Web services is proposed. The chapter details how dependability of composite Web

services is achieved by controlling concurrency and encapsulation of erroneous states within nested actions and by a forward error recovery mechanism based on coordinated exception handling. Chapter V presents the execution platform and associated middleware services. Finally, Chapter VI concludes this thesis with the summary of our contribution and the perspectives for our work.

II Background

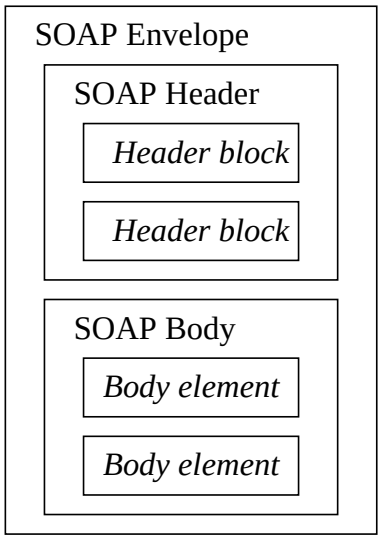
The Service-Oriented Architecture (SOA) approach appears to be a convenient architectural style towards developing complex systems by composing autonomous applications deployed on the Internet. In this context, the Web service architecture has become the major enabling technology for building service-oriented computing systems. This chapter presents the main elements of the Web services architecture and the related work on the composition of Web services and on providing dependability.

II.1 The Web services architecture

Standardization organizations such as the W3C[1] and OASIS[2] have defined various specifications for describing the elements of the Web service architecture. The XML text format [W3C, 2004a] is adopted for describing the specifications and for encoding information exchanged between systems. The main reason for choosing XML is its simplicity and ease of adaptation to any platform and to the main application protocols of the Internet. In conformity with the SOA architectural style [Papazoglou and Georgakopoulos, 2003], Web service providers and service requesters communicate by exchanging messages over an Internet application protocol. Interfaces of Web services are described using a language-neutral document. Web services are further localized by service requesters using some discovery protocol. By providing standardized, platform neutral, message-oriented communications using standard Internet protocol, interface definition languages, and service discovery support, Web services enable building service-oriented systems on the Internet, allowing the inter-operation and the composition of autonomous systems across heterogeneous platforms. In the following,

[1] World Wide Web Consortium, `http://www.w3.org`

[2] Organization for the Advancement of Structured Information Standards, `http://www.oasis-open.org`




```
<env:Envelope xmlns:env="http://
    www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://
        example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
  <m:alert xmlns:m="http://
    example.org/alert">
      <m:msg>Alert message</m:msg>
  </m:alert>
  </env:Body>
</env:Envelope>
```

Figure II.1: SOAP document structure

the specifications proposed by the aforementioned standardization organisms related to messaging, description and discovery are presented with an emphasis on the issues for expressing composability and dependability of provided services.

II.1.1 Messaging

Messaging between Web services and service requesters is realized by encoding data using the SOAP messaging protocol [W3C, 2003b]. The SOAP specification defines a protocol for information exchange that sets the rules of how to encode service requests, responses and data in XML. In addition, various bindings of SOAP messages to Internet application-level transport protocols are provided. The structure of a SOAP message is simple with little syntactic requirements on message contents, but can be extended with additional specifications for adding new features such as headers for security, reliability, or correlation information for tracking multiple messages. The structure of an XML-encoded SOAP message is represented in Figure II.1, together with an example of SOAP message. A SOAP message consists of an outermost *Envelope* XML element with two sub-elements: the *Header* element, which comprises zero or more header blocks, and the *Body* element comprising zero or more element information items.

SOAP further enables expressing the capabilities of service requesters and service providers (called SOAP nodes) using SOAP *features*. A capability is identified by an URI and can be used by another SOAP node, which understands this capability, to realize the interaction. One such a capability is the *Message Exchange Pattern* (MEP), which specifies the messaging behaviour supported by a SOAP node. Two MEPs are defined in the SOAP v1.2 specification: *request-response* and *response*. The request-response MEP defines an RPC-like communication pattern by defining a request SOAP message and a subsequent response SOAP message. The response MEP defines a SOAP message sent as a response to a received non-SOAP request message. The underlying protocol used for communicating SOAP messages should be able to realize asynchronous communications, as defined in the related MEPs, and transfer XML-based messages. Any Internet application protocol may be used for exchanging SOAP messages as long as it provides these features. HTTP [IETF, 1999] is the most common protocol that is widely used, due mostly to its widespread availability on various platforms and its simplicity. However, other underlying protocols such as SMTP [IETF, 1982] can also be used.

A first requirement for dependability in Web services is to guarantee reliable interactions between service requesters and providers. Underlying Internet transfer protocols used for communicating SOAP messages guarantee reliable delivery of message contents. However, they are not sufficient for addressing reliability at the application-level. Indeed, interactions between service providers and requesters are done by exchanging a sequence of different messages, in a specific order. Asynchronous communications can lead to message losts that can not always be notified to the sender of the message. Moreover, duplicate messages can be sent by one party if it considers that the initial message have not been received, which might not be the case. Based on established solutions for addressing reliable application communication in distributed systems, several protocols have been introduced as a complement of the messaging protocol of Web services. Reliable messaging protocols for Web services address the guaranteed delivery of messages and the elimination of duplicated messages. WS-Reliability [OASIS, 2004b] and WS-ReliableMessaging [BEA Systems et al., 2004] protocols define special headers for SOAP messages for identifying messages and propose acknowledgment-based protocols. These specifications are independent of the underlying protocol although concrete bindings for using them over HTTP are provided. Alternatively, the HTTPR specification defines a reliable messaging protocol and extends HTTP headers with reliability attributes [IBM, 2002]. It is worth noting that two SOAP nodes should support the same reliable messaging protocol in order to use it. SOAP *features* are used for identifying the supported protocol and for defining the information on how to implement it.

Ensuring reliable messaging using acknowledgment protocols and message oriented middlewares have been widely used in distributed systems. The specific protocols introduced for Web services take advantages of these solutions and are already integrated in current products[3]. We believe that these protocols and their implementation address the issue of reliable messaging in Web services. In the remainder, we assume the reliability of single interactions with a Web service, using one of the above protocols.

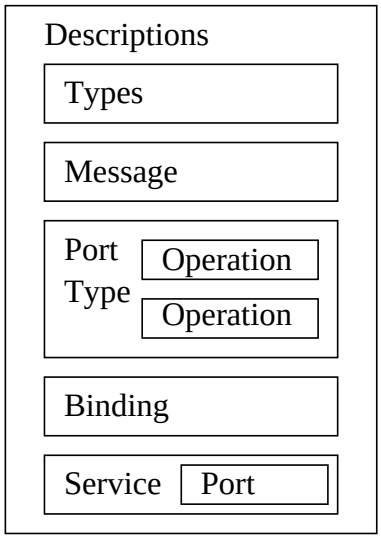
II.1.2 Description

WSDL is the language recommended by the W3C for describing Web services [W3C, 2005]. WSDL provides means for specifying the functional interface of Web services, similarly to existing interface definition languages like CORBA IDL [OMG, 2002]. However, WSDL further enables the specification of concrete details needed for accessing the service such as the network end-point address and the supported message encoding format and communication protocols (e.g., SOAP over HTTP). A service requester should be able to interact with the Web service, based only on this interface definition. The WSDL language provides base constructs for describing Web services, but is also extensible to allow expressing a variety of additional information.

A Web service is described by a set of operations, which can be, in general, called independently of each other. Operations are further associated to a set of messages that can be of three types: *input* messages for specifying the reception of a message, *output* messages for specifying messages sent by the Web service (in general as a response to an input message), and *fault* messages for specifying error messages sent to service requesters. Messages can be constituted of several typed parameters. The WSDL specification allows using any type system for defining the types of the parameters. The type system that is used is identified in the WSDL description using an URI. Data types of XML Schema, which is the W3C specification for describing a class of XML documents [W3C, 2004c], is the common choice.

The general structure of a WSDL (v1.1) document is depicted on the left-hand side of Figure II.2, with an excerpt of a WSDL document related to a calculator

[3] At the time of this writing, Apache Axis2 (`http://ws.apache.org/axis2/`), Oracle BPEL Process Manager (http://www.oracle.com/technology/products/ias/bpel/index.html) and Microsoft WSE 3.0 (`http://msdn.microsoft.com/webservices/webservices/building/wse/`) support the WS-ReliableMessaging protocol. RM4GS (`http://businessgrid.ipa.go.jp/rm4gs/`) implements the WS-Reliability protocol.




```
<message name="addRequest">
  <part name="i" type="xsd:int"/>
  <part name="j" type="xsd:int"/>
</message>
<message name="addResponse">
  <part name="return" type="xsd:int"/>
</message>
<portType name="Add">
  <operation name="add">
    <input message="addRequest"/>
    <output message="addResponse"/>
  </operation>
</portType>
<binding name="CSoapBinding style="rpc" transport=
  "http://schemas.xmlsoap.org/soap/http"/>
  <operation name="add">
  <operation soapAction=""/>
    <input><body use="encoded" ... /></input>
    <output><body use="encoded" .../></output>
  </operation>
</binding>
<service name="AddService">
  <port name="Add" binding="CSoapBinding">
    <address location="http://localhost:81/C"/>
  </port>
</service>
```

Figure II.2: WSDL v1.1 document structure

Web service definig the *add* operation. Specifically, a WSDL document consists of a set of elements composing the abstract part, and a set of elements composing the concrete part, all of them contained in a top-level *definitions* element. The abstract part consists of the XML elements *types* for declaring internal data types, *message* for declaring sent and received messages and *portType*, which comprises a set of *operation* elements defining the operations. To each operation defined in a WSDL document, a set of properties may be associated by giving the URIs of the desired properties. In particular, in WSDL v2.0, the message exchange pattern supported by a Web service operation is described by setting the *pattern* property of the operation to the unique identifier of the MEP[4]. For instance, for describing that an operation implements the request-response MEP, the *pattern* property of the operation should be set to `http://www.w3.org/`

[4]In WSDL v1.1, the MEP is derived from the declaration order of input and output messages

`2005/05/wsdl/in-out`. The concrete part consists of XML elements *binding* for defining the communication protocols and the message formats used by the Web service, and *service* that specifies the Web service name and a set of *ports*, which define *endpoints* by associating a binding to a network address.

The exceptional behaviour of a Web service is in part given by its WSDL document. WSDL declares fault messages that are sent or received when the normal flow of messages is disrupted during the execution of a message exchange. Propagation of fault messages are also defined using MEPs. WSDL v2.0 defines three MEPs for specifying fault propagation: the *fault replaces message* pattern, the *message triggers fault* pattern and the *no faults* pattern. The first pattern specifies that any message can be substituted by a fault message. For example, when an error occurs, which prevents the emission of a message, a message indicating the failure can be sent instead. The second one specifies that a message that has already been sent may trigger a fault message. The fault message must be sent to the originator or the message that triggered it. The last pattern specifies that no fault message should be sent during the interaction.

Summarizing, WSDL is used to describe the operations provided by a Web service and the messaging behaviour of each operation individually. However, interactions with Web services often implies calling several operations on the same Web service. Moreover, operations of a single Web service may need to be called in a specific sequence for realizing a specific task. For some Web services, following a specific order can also be mandatory for getting the expected result from the Web service. For example, a Web service may require from a service requester to call first a specific operation for authentication before calling any other subsequent operations. We believe that for composability, the description of the ordering requirements of the operations of a Web service (called conversations) must be part of the service interface. Specification of the supported interaction protocol of a Web service is actually not addressed with WSDL. Several languages complementing WSDL for addressing this issue have been proposed; they are surveyed in Section II.2.2.

II.1.3 Discovery

Complementary to the above core Web services architecture elements, is the UDDI (Universal Description, Discovery and Integration) standard, which specifies a registry for dynamically locating and advertising Web services [OASIS, 2004c]. UDDI servers were designed initially as centralized registries, but the specifications allow registries to organize themselves into more complex networks

for increasing availability, reducing system load or for providing specialized directories. Service requesters make queries to UDDI registries, using a standard service discovery interface.

Web service registries and associated service discovery protocols play an important role in service composition and in providing dependability. Indeed, composite services can be designed by integrating composed Web services that are only described abstractly. By abstract, we mean that composed Web services are identified in a composite Web service with their abstract interfaces, but with no concrete details about the localization and name of the service. Web service instances that provides the same (or compatible) interfaces can then be discovered prior or during execution by querying a service registry. In particular, if a Web service becomes unavailable during the execution of an interaction, the service requester can query a UDDI registry to find an alternative Web service. Then, depending on the application semantics, the interaction can be continued from where it was interrupted, or be retried from the beginning.

II.2 Web service composition

When designing a composite Web service, developers should answer several questions: *Which Web service can be integrated ? What is the interaction protocol that should be implemented ? How to express the control flow of the composite Web service ?* Three complementary aspects of composability in the Web services architecture have been identified for giving answers to these questions: *conversations*, *choreography* and *orchestration*.

(1) *Conversations* address the first issue by giving the interactions that an individual Web service supports. Conversations are described, in particular, in terms of ordering requirements over the operations that a Web service provides.

(2) *Choreography* gives an answer to the second question by specifying the interaction protocol for a specific business process involving several Web services.

(3) *Orchestration* address the third issue by providing means for specifying the composition process.

Several languages for specifying the above aspects have been proposed. Although a clear categorization of proposed languages into these aspects is not always pos-

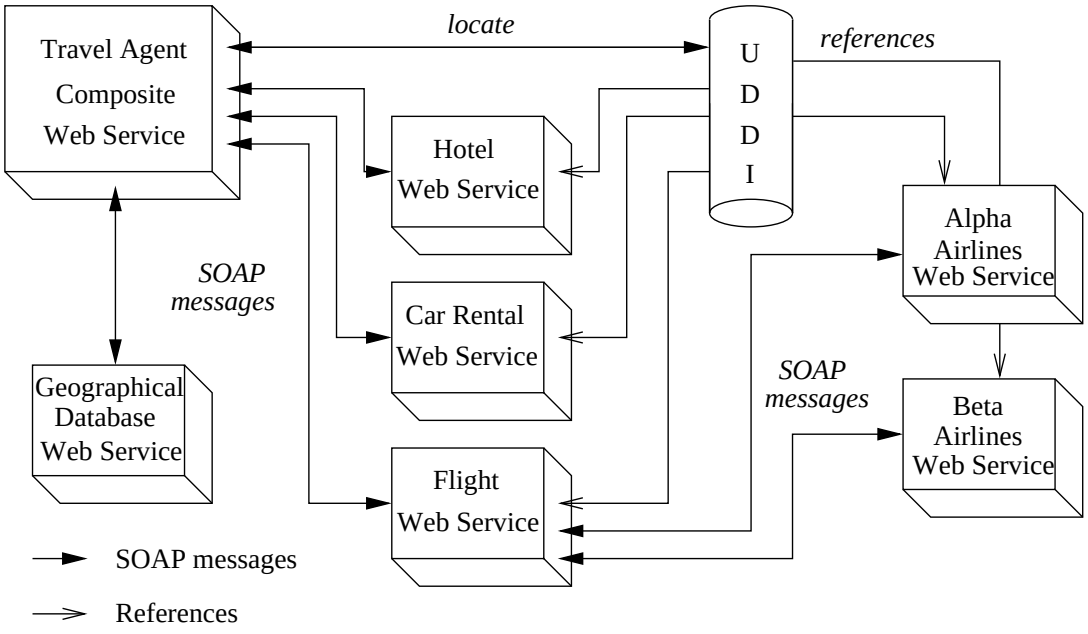


Figure II.3: A composite Web service example: the travel agent service

sible as they offer overlapping features, we survey in this section proposed languages and their support for expressing dependability. Indeed, building dependable composite services needs specifying dependability properties in all the above aspects. In particular, conversation languages should allow expressing the exceptional behaviour of individual Web services. The choreography should enable expressing different recovery protocols and an orchestration languages should offer constructs for expressing dependability mechanisms. First, we present a use case to illustrate a composite Web service with dependability requirements. Then, the above aspects are considered separately and exemplified.

II.2.1 A use case

Figure II.3 depicts the travel agency case study application introduced in Section I, which we use throughout this thesis to exemplify various aspects of Web service composition. A trip is composed of a transportation option to get to and from the destination and an accommodation reservation for the duration of the journey. Additional services may complement the trip offer such as travel insurances or car rental services. Furthermore, third party services can be used to help the system to find appropriate choices such as geographical database systems for locating cities and computing distances or services for converting units

and finding correct airport codes for the destination. The travel agency interacts with several existing independent Web services located through a public Web services registry. Note that each of the composed Web service may be itself a composite service, as shown in the example below with the *Flight* Web service which composes Web services of different airline companies. Typical challenges that the designer of such a composite application faces are related to: (i) the coordination of autonomous and concurrently running components for providing an integrated service, and (ii), ensuring a high level of dependency at the composite application level despite the use of potentially undependable and autonomous composed services. In particular, the travel agent may have to deal with failures occurring during the composition such as partially completed trip reservations, transparently to the requester of the composite Web service.

II.2.2 Conversations

As raised earlier in Section II.1.2, operations of a Web service are described in the associated WSDL document, which gives the messaging pattern of each operation. However, to correctly use a Web service and get the expected result, a service requester needs to know, in addition, the order in which it should call the operations offered by the Web service. Since an agreed terminology does not exist yet in the Web services community for defining ordering requirements over the operations of a Web service, we use the term *conversation* as it is introduced in the Web Services Conversation Language (WSCL): *a conversation specifies the XML documents being exchanged, and the allowed sequencing of these document exchanges* [W3C, 2002b]. Conversation languages define the conversations that are supported by a Web service. Conversations enable thus defining a higher level of interaction protocol by expressing ordering dependencies between operations, in a way similar to *path expressions* introduced for the synchronization of concurrent processes [Campbell and Habermann, 1974]. A conversation is considered as an extension of the abstract interface of a Web service, complementing the WSDL definition.

Various conversation languages for Web services have been introduced in the literature, which may be coupled or not with the specification of Web services composition. Solutions may be distinguished according to whether the conversation is expressed from the point of view of an external observer, or if it is given from the point of view of an individual service. The first approach enables in particular to link conversations supported by different Web services for expressing multi-party conversations, as in the language introduced in CS-WS [Hanson et al., 2002]. However, this approach limits the expressiveness of a ser-

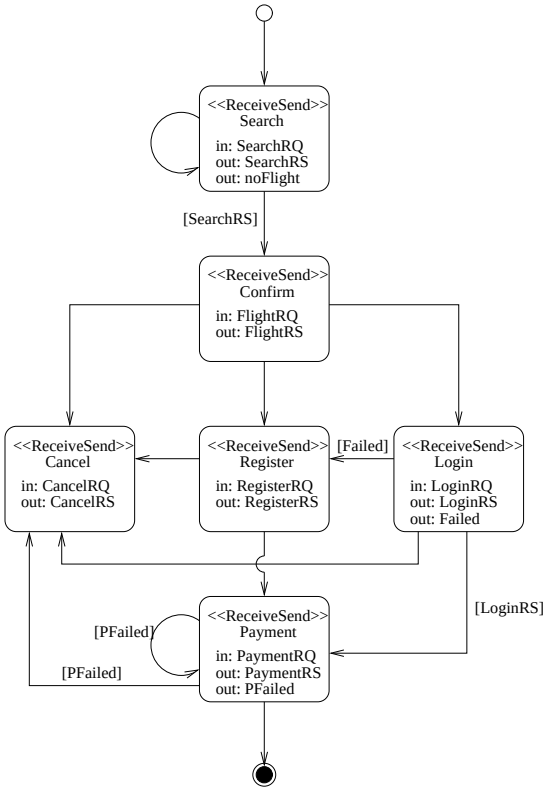
vice interface by introducing a tight coupling between conversations of different Web services. The second approach, which is adopted by most of the proposed conversation languages, enables defining conversation support independently of any specific interaction protocol, complementing directly the WSDL definition. Languages proposed for describing conversations of Web services from an individual Web service perspective include DML [Tolksdorf, 2003], WSCL [W3C, 2002b, Frolund and Govindarajan, 2003], the framework introduced in [Benatallah et al., 2004], the service model description of OWL-S [W3C, 2003a], the conversation specification presented in [Yi and Kochut, 2004], WSCI [W3C, 2002a] and the service specification language introduced in [Jimenez-Peris et al., 2003].

Conversations of Web services that are expressed in the above languages are generally represented using state transition diagrams. In particular, WSCL uses the UML activity diagram for modeling conversations where activities represent the operations that may be called, and transitions the execution of operations. Labels on transitions representing an output message are used to set conditions on the transition. Figure II.4 illustrates the conversation of a flight Web service expressed in WSCL together with an associated UML activity diagram. The conversation states that the interaction with the Web service starts when a requester calls the *Search* operation, which may be called indefinitely. If the operation returns a valid response (the *SearchRS* output message), then the requester is allowed to call the *confirm* operation, by selecting the flight that it wants to book. Then, the requester can *cancel* the reservation or *login* to proceed to the payment. If it can not login, because it never registered before, the requester can call the register operation to create an account. If the login fails, the requester can retry or register. Once the login has been made, either by registering a new account or by calling the login operation, the payment can be done by calling the *payment* operation. Again, if the payment fails, the requester can retry or cancel the flight reservation. An XML excerpt of the conversation is also given on the right-hand side of the figure, where the *ReceiveSend* interaction type represents a request-response MEP.

Providing machine-readable specifications of conversations in the service interface is beneficial for at least three reasons:

(1) Conversations are used in the discovery process where conversations supported by Web services are matched to conversations implemented by service requesters.

(2) Specification of conversations of a Web service are used to check correctness of interactions engaged by a service requester, prior or during execution.




```
<Interaction interactionType="ReceiveSend"
    id="Login">
  <InboundXMLDocument id="LoginRQ"
    hrefSchema="http://foo.org/LoginRQ.xsd">
  </InboundXMLDocument>
  <OutboundXMLDocument id="LoginRS"
    hrefSchema="http://foo.org/LoginRS.xsd">
  </OutboundXMLDocument>
  <OutboundXMLDocument id="Failed"
    hrefSchema="http://foo.org/Failed.xsd">
  </OutboundXMLDocument>
</Interaction>

<Transition>
  <SourceInteraction href="Login"/>
  <DestinationInteraction href="Register"/>
  <SourceInteractionCondition href="Failed"/>
</Transition>
```

Figure II.4: WSCL-based conversation of the flight Web service

(3) Conversations are used to automate the implementation of service compositions, e.g., by providing tools for the automated generation of correct code skeletons [Meredith and Bjorg, 2003, Benatallah et al., 2004].

The first and second issues above require ways for verifying the compatibility of different conversations. Given the process algebra specification of a conversation language, behavioral compatibility of Web service clients (e.g., a composite Web service when accessing composed Web services) with Web services can be verified using observational equivalence relations. Furthermore, if a Web service defines all the conversations that it supports, and a service requester searches for a Web service, which supports one specific conversation, then compatibility of the Web service can be verified if there is a simulation pre-order relation between the two conversations. A conversation simulates another conversation if it can match all of its moves. Specifically, the conversation of the Web service must simulate the required conversation of the service requester.

Using conversations for specifying the error recovery behaviour of a Web service is important for implementing an efficient dependability mechanism. For exam-

ple, using conversations, a Web service can specify which sequence of operations should be called for compensating the effects of a previously called operation, or sequence of operations. Existing languages mentioned previously provide no or limited support for describing recovery behavior of Web services. These languages mainly allow exceptional behavior to be described using transitions on fault messages. Hence, except the framework introduced in [Benatallah et al., 2004], specification of recovery properties such as transactional behavior of conversations is not addressed. The language introduced in [Benatallah et al., 2004] allows specifying transactional behavior of conversations. However, it uses a list of pre-defined transactional properties hence reducing the language's expressiveness. Furthermore, existing conversation languages do not address timing issues, except for CS-WS [Hanson et al., 2002] that introduces an additional timeout attribute associated to operations. Also, it should be possible to specify concurrent activities within conversations, as concurrency allows specifying complex distributed systems involving several competing and/or collaborating participants. However, only the workflow-based WSCI language address the specification of concurrent abstract processes [W3C, 2002a].

II.2.3 Choreography

A *choreography* describes the message exchange rules among multiple interacting parties. Concretely, it is achieved by linking sent and received messages of different parties and specifying a control flow among related message exchanges. If conversation supports for all services already exist, a choreography can also be specified by associating received messages of one service with a sent message of another. It is worth noting that the same conversations can be linked in different ways to specify different choreographies. Conversations are thus used to describe all supported interactions of a Web service, while a choreography gives the interaction protocol for a specific composite task involving several Web services and service requesters.

A choreography can be instantiated by assigning concrete Web services to the different roles defined in the protocol. Therefore, if a Web service describes its supported conversation, the conversation should be compatible with the part of the choreography defining the interactions with it. Moreover, different roles defined in a choreography can be bound to a single Web service. Defining a service composition through a choreography suggests a top-down development approach where the composition will initially be described abstractly without identifying any Web service. Then, composed Web services will be built according to the definition of choreography role, or matching existing Web services will be found

and integrated in the composition.

For easing the development of dependable composite Web services, choreographies can be used for:

(1) Specifying transaction protocols and/or transactional behaviour.

(2) Specifying exceptional behaviour that must be executed when an exceptional condition occurs, in which several roles can be involved.

(3) Verifying at design-time or at run-time, conformity of the conversations supported by Web services and of their implementations with the behaviour of the associated roles [Foster et al., 2004, Fu et al., 2004, Martens, 2005].

Existing languages for specifying choreographies include WS-CDL from the W3C choreography working group [W3C, 2004b], which specifies interaction protocols by defining a global control flow among interacting parties using explicit control structures. The model presented in [Bultan et al., 2003] defines choreographies (referred to as conversations by the authors) using Mealy machines [Mealy, 1955]. In addition, the conversation languages CS-WS [Hanson et al., 2002] and WSCI [W3C, 2002a] introduced in the previous section, enable defining choreographies by linking conversations of different Web services.

The sequence diagram in Figure II.5 illustrates a choreography for the travel agent service. The choreography involves five roles, a *customer*, a *travel agent*, an *airline*, an *hotel* and a *bank*. The customer interacts with the travel agent for booking a trip and with the bank for making the payment. The travel agent interacts with the *hotel* and *airline* services for reserving flights and rooms, with the *bank* to make the payments of reserved flights and rooms on behalf of the customer, and with the customer, to which it sends the confirmation for the trip. The bank service further interacts with the airline and hotel services for confirming payments. The roles represent abstract services for which corresponding concrete Web service instances should be assigned in order to execute the travel organization task defined by the choreography. Does the flight reservation Web service introduced in Section II.2.2, and which supports the conversation depicted in Figure II.4, can be integrated in the above choreography ? At a first glance, it seems that it is not possible, because the conversation of the flight reservation Web service requires that the payment should be made by calling an operation of the Web service. However, in a choreography, a single Web service may also be assigned to more than one role, at different parts of the interaction protocol. For example, an airline Web service that supports both booking and

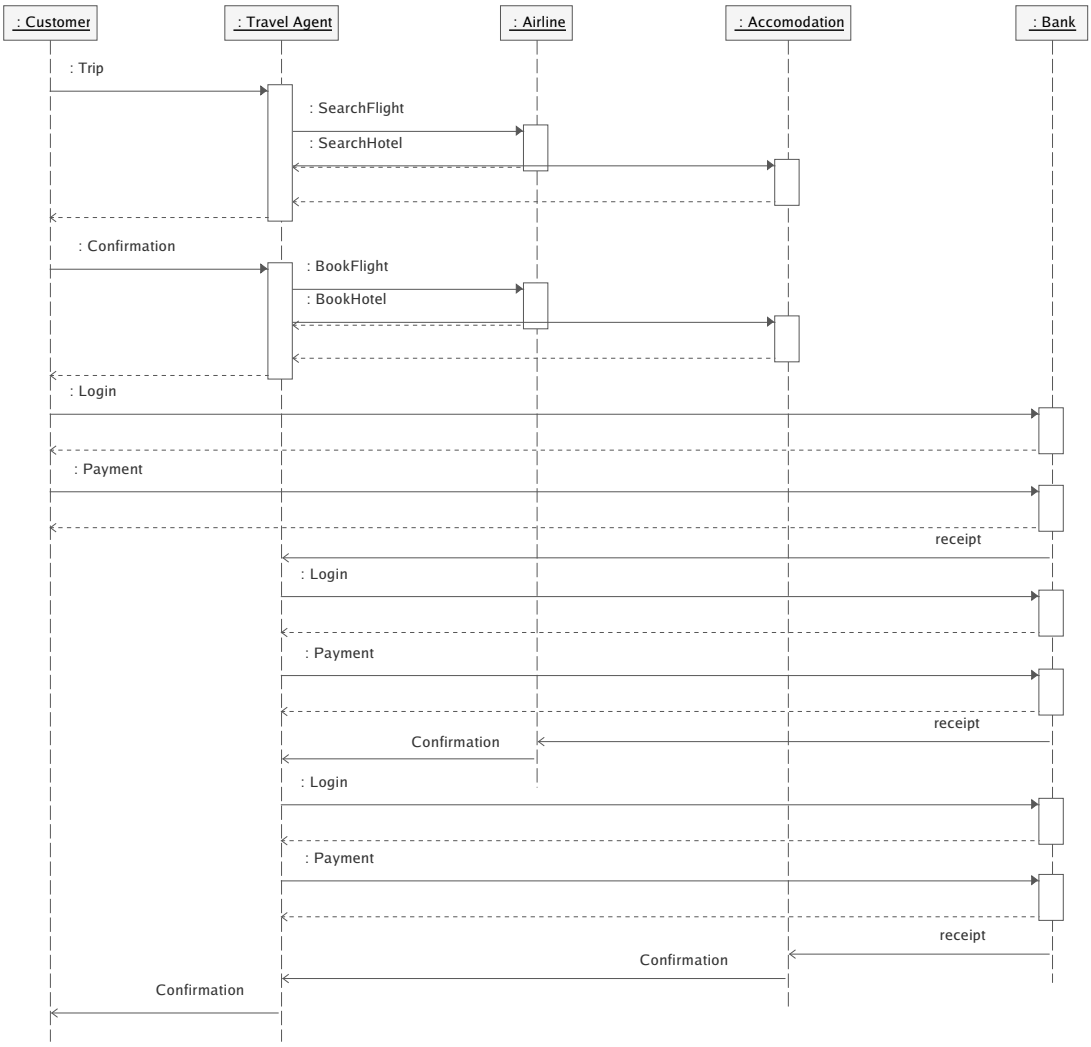


Figure II.5: Choreography for the travel agency service composition

on-line payments can play the roles of the *airline* and of the *bank* for making the payment of the booked flight. Furthermore, different Web services can also be assigned for a single role. For example, different bank Web services can be used for making the three different payments. Indeed, if we consider an abstract Web service assigned to the *airline* and *bank* roles of the choreography, the required conversation would be: an operation for searching flight, followed by a confirmation, and then the login and the payment. This conversation can easily be simulated by the conversation of the flight Web service.

II.2.4 Orchestration

The term *orchestration* refers to an executable workflow process, which interacts with external Web services. Orchestration can thus be used to specify a composite Web service. It specifies the control flow of the composition, from the point of view of the composite service. In particular, it can be used to implement different roles of a choreography, which contrary to orchestration, describes the interactions of all parties involved in the composition. Orchestration is distinguished from conversation and choreography by the fact that it specifies the internal mechanisms of a Web service, and need thus not to be publicly available nor shared between interacting parties. However, the specification can be used to derive the conversations supported by the composite Web service and to verify its compatibility with a choreography.

The composition process can be specified as a graph (or process schema) over the set of composed Web services, defining the invocation order of composed Web service operations. Several models for specifying orchestrations, based on existing solutions for the coordination of distributed activities, have been proposed. State-charts are used to specify composite Web services in the Self-Serv environment, where composed services are coordinated by peer-to-peer interactions [Benatallah et al., 2005]. Another approach is to model the composition based on Petri nets [Reisig and Rozenberg, 1998], as addressed in [Narayanan and McIlraith, 2002] and [Hamadi and Benatallah, 2003]. BPML [A. Arkin, 2002] and BPEL [BEA Systems et al., 2005] are workflow specification languages, which are based on process algebra. Formalization of BPEL processes has indeed been the subject of many publications [Farahbod et al., 2004, Schmidt and Stahl, 2004, Foster et al., 2004, Fu et al., 2004, Ferrara, 2004]. Automated composition of Web services is also considered [Narayanan and McIlraith, 2002, Medjahed et al., 2003]. Automatic composition is attractive but restricts the composition patterns that may be applied, and cannot thus be used in general. Approaches that introduce XML-based declarative languages for specifying workflow processes directly support reuse, openness, and evolution of Web services by clearly distinguishing the specification of component Web services (comprising primitive components that are considered as black-box components and/or inner composite components) from the specification of composition. Hence, although there is not yet a consensus about the best approach for specifying composite Web services, it may be anticipated that this will most likely rely on the XML-based specification of a graph over Web services that is decoupled from the specification of the composed Web services. The main reasons that lead to this conclusion include compliance and complementarity with established W3C standards, providing reusability, openness and extensibility, but also the fact that it is the

approach undertaken by most industrial consortia.

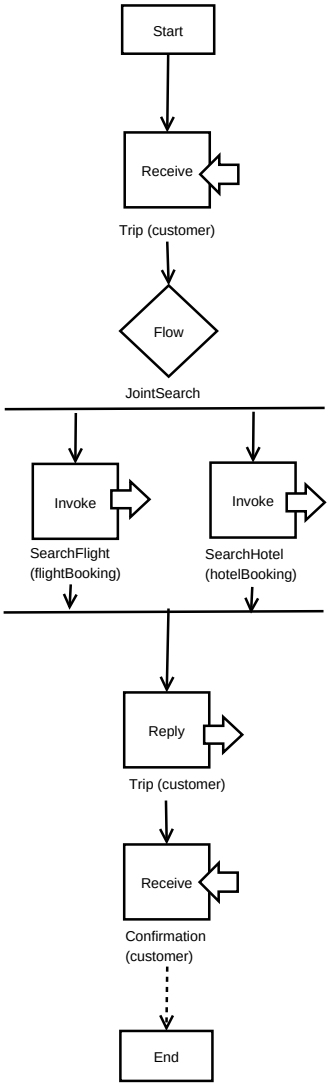


Figure II.6: BPEL example for the the travel agent

BPEL in particular is the most popular orchestration language. It is developed by merging two prior proposals: XLANG [Thatte, 2001], which is based on the π-calculus process algebra [Milner, 1999] and WSFL [F. Leymann, 2001], which is based on a Petri net model [Reisig and Rozenberg, 1998], and is submitted for standardization to the OASIS consortium. As such, several workflow engines executing BPEL processes have been developed. For illustrating an orchestration, the BPEL specification of a composite Web service, implementing the role *Travel Agent* in the choreography illustrated in Figure II.5 is given in Figure II.6.

II.3 Fault tolerance in the Web services architecture

The specifics of Web services and of their composition require special care in the design of supporting fault tolerance mechanisms, which is the focus of this section. Both backward and forward error recovery mechanisms for the Web services architecture are considered.

II.3.1 Fault tolerance mechanisms

In general, the choice of fault tolerance mechanisms to be exploited for the development of dependable systems depends very much on the fault assumptions and on the system's characteristics and requirements. There are two main classes of error recovery [Lee and Anderson, 1990]: backward error recovery, which is based on rolling system components back to some previous correct state, and forward error recovery, which consists of transforming the state of the system components into any correct state. It is a widely-accepted fact that the most beneficial way of applying fault tolerance is by associating its measures with system structuring units as this decreases system complexity and makes it easier for developers to apply fault tolerance [Randell, 1983]. Structuring units applied for both building distributed systems and providing their fault tolerance are well-known: they are *distributed transactions* and *atomic actions* [5]. Distributed transactions [Gray and Reuter, 1993] use backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the ACID (atomicity, consistency, isolation, durability) properties. Atomic actions [Campbell and Randell, 1986] allow applying both backward and forward error recovery. The latter relies on coordinated handling of exceptions that involves all action participants. Backward error recovery has a limited applicability, and in spite of all its advantages, modern systems are increasingly relying on forward error recovery, which uses appropriate exception handling techniques as a means [Cristian, 1989]. Examples of such applications are complex systems involving human beings, COTS components, external devices, several organizations, movement of goods, operations on the environment, real-time systems that do not have time to go back. Composite Web services clearly fall into this category.

As an example composite Web service that shows the need for a specialized fault tolerance mechanism, consider the following scenario with the previously introduced travel agency composite Web service. For a given trip request, the

[5] also referred to as conversations, but we will not use this term to avoid confusion with Web services conversations.

travel agent finds, by querying composed Web services providing hotel and flight reservation systems, a list of available hotel rooms and flight schedules. The service requester makes its choice by selecting the cheapest trip and confirms booking. The travel agent then attempts for the bookings accessing respective composed Web services. Interactions with the hotel reservation system terminates normally, and the hotel room is booked. However, the flight reservation fails for some reason (e.g., flight no more available at the requested price), and the complete trip reservation cannot be completed. Typical solutions that can be applied for dealing with this uncompleted transaction are:

(1) Backward error recovery by restoring the state back: the hotel reservation is cancelled –if possible. The service requester can then retry a new reservation from the beginning as no reservation has been kept.

(2) Forward error recovery: the reservation can be retried by the travel agent on an alternative flight reservation system. If the second booking attempt succeeds, then the reservation process continues normally by confirming it to the service requester. Alternatively, if no flight can be scheduled and the hotel reservation can not be cancelled, the travel agent can propose other transportation means to the service requester.

It is worth noting that backward and forward error recovery are complementary, and complex applications often use both means for achieving dependability. Developing fault tolerant mechanisms for composite Web services has been an active area of research. Existing proposals mainly exploit backward error recovery, and more specifically flexible transactional models introduced in the context of multi-database systems [Elmagarmid, 1992]. However, the lack of transactional support of autonomous Web services has led to exploit complementary forward error recovery techniques.

II.3.2 Backward error recovery for the Web

Transactions [Gray and Reuter, 1993] are widely used in database management systems for providing reliability and consistency in the presence of concurrency (concurrent updates on the same data item) and of failures during database operations. A transaction is defined as a unit of computation that must guarantee four properties: *atomicity*, *consistency*, *integrity*, and *durability* (also referred to as ACID properties). The *atomicity* property ensures that the unit of computation is executed up to completion or not at all. *Consistency* ensures that

a transaction, once completed, changes the system from one consistent state to another consistent state. The *isolation* property ensures that intermediate values of a transaction are not seen from the outside and in particular from concurrently running transactions. Finally, *durability* is the property that guarantees that once a transaction has committed, its results are permanently recorded and persist to subsequent system failures. Transactions have been proven successful in enforcing dependability in closed distributed systems and are extensively exploited for the implementation of primitive (non-composite) Web services. Enforcing ACID properties typically requires introducing protocols for: (i) locking resources (e.g., two-phase locking protocols) that are accessed for the duration of the embedding transaction, and (ii) committing transactions (e.g., two- or three-phase commit protocols). However, transactions are not suited for making the composition of Web services fault tolerant in general, for at least two reasons:

- The management of transactions that are distributed over Web services requires cooperation among the transactional supports of individual Web services, which may not be compliant with each other and may not be willing to do so, given their intrinsic autonomy and the fact that they span different administrative domains.
- Locking resources (i.e., the Web service itself in the most general case) until the termination of the embedding transaction is in general not appropriate for Web services, still due to their autonomy, and also to the fact that they potentially interact with a large number of concurrent service requesters that will not stand extensive delays.

Enhanced transactional models have been considered to alleviate the latter shortcoming. In particular, open-nested transactions [Pu et al., 1988, Garcia-Molina and Salem, 1987], where transactions may be composed of a number of concurrent nested transactions that can commit independently relax the *isolation* property allowing partial results to be seen outside of nested and top-level transactions. Open nested transactions are more suited for long running transactions, reducing in particular latency due to locking. Typically, open-nested transactions are matched to the transactions already supported by Web services. Aborting the whole transaction requires using compensation over committed nested transactions, which consists of running nested transactions in order to undo effects of committed nested transactions. Because effects of previously committed nested transactions can be read by external transactions, the atomicity is said *semantic* [Garcia-Molina, 1983]. It does not guarantee that the state will be rolled back, but that a semantically equivalent state can be reached. Semantic atomicity

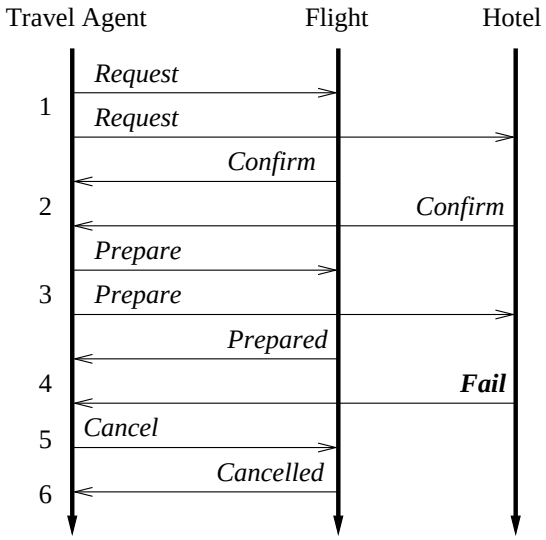
is ensured if all nested transactions either commit or compensate. For executing Web service operations in an open-nested transaction, Web services should provide compensating operations for all the operations they offer. This is in particular addressed by the BPEL [BEA Systems et al., 2005] and WSCI [W3C, 2002a] composition languages, which allow defining compensating operations associated with the Web services operations. It is worth noting that when several Web services are involved in an open-nested transaction, effects of the compensated nested transactions on all Web services must be compensated as well (i.e., cascading compensation by analogy with cascading abort), which requires in addition the coordination of compensating operations on each Web service. Several solutions following two different approaches are being proposed in the context of Web services for coordinating open-distributed transactions.

The first approach enforces participant Web services to describe their supported transactional behaviors. Then, a service requester, or a middleware service acting on behalf of the service requester, exploits those descriptions for specifying and executing a (open-nested) transaction over a set of Web services. The termination of the transaction is dictated by the outcomes of the transactional operations invoked on the individual services. Such a concern is addressed in the WSTx framework [Mikalsen et al., 2002, Tai et al., 2004] and in the WebTransact framework [Pires et al., 2003b,a]. However, the transactional behaviour is described at the operation-basis, which is not sufficient for comprehensively expressing the recovery behaviour of a service. Indeed, the recovery strategy to be implemented by the client may involve calling more than one operation for a single transactional behaviour (e.g., calling a sequence of operations in a specific order for cancelling a transaction). Such a feature is addressed in [Benatallah et al., 2004], which introduces an XML-based language that allows annotating conversations with non-functional properties. Nevertheless, the transactional protocols that can be applied is inevitably limited and require knowledge of the precise semantics of the transactional behaviours, which are expressed using meta-data.

In addition to the above client-side solutions to the coordination of distributed open-nested transactions, work is undertaken in the area of distributed transaction protocols supporting the deployment of transactions over the Web, while not imposing long-lived locks over Web resources.

The Business Transaction Protocol (BTP) introduces two different transaction models for the Web: (i) the *atomic business transactions* (or *atoms*), and (ii) the *cohesive business transactions* (or *cohesions*) [OASIS, 2004a]. A composite application can be built from both *atoms* and *cohesions* that can be nested. In the *atomic business transaction* model, several processes are executed within a transaction and either all complete or all fail. This is similar to distributed

ACID transactions on tightly coupled systems. However, the isolation property is relaxed and intermediate committed values can be seen by external systems (i.e., systems not enrolled in the transaction). Figure II.7 illustrates the *atomic business transaction* model using the travel agent service involving a flight booking Web service (*Flight*) and an accommodation booking Web service (*Hotel*). In this scenario, the hotel room booking fails while the flight booking succeeds, which leads to cancellation of the booked flight before the end of the transaction. The *cohesive business transaction* model allows non-ACID transactions to be defined by not requiring successful termination of all the transaction's participants for committing. A travel agent service scenario example for illustrating *cohesive business transactions* is given in Figure II.8, where the flight booking is performed on two distinct Web services. In this example, the transaction, which was originally initiated with three participants, ends with two commits and one abortion.



(1) *Travel Agent* sends the request messages to *Flight* and to *Hotel* Web services.

(2) *Flight* and *Hotel* respond (*Confirm* messages) with listings of available flights and hotel rooms.

(3) *Travel Agent* orders the bookings by initiating commitments (*Prepare* messages).

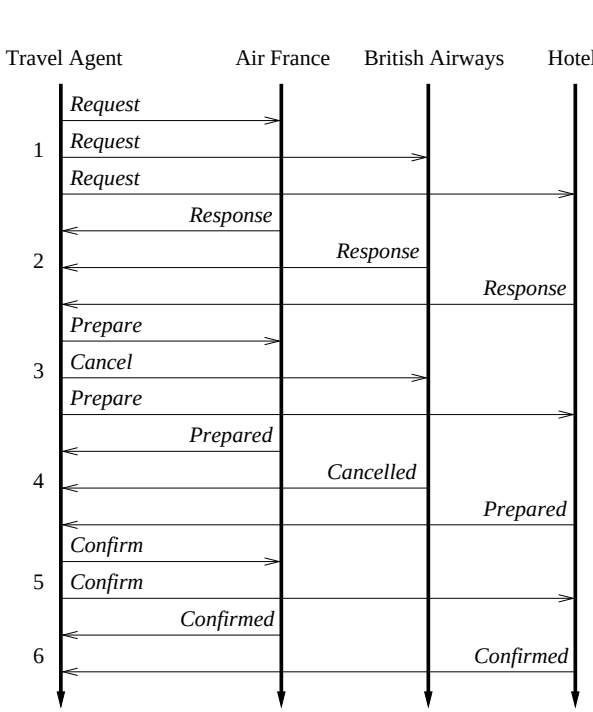
(4) *Flight* Web service returns *Prepared* and is ready to commit, while the *Hotel* Web service returns a *Fail* error message. Commit is no longer possible on the *Hotel* Web service for this transaction.

(5) *Travel Agent* cancels the transaction on the *Flight* Web service by sending the *Cancel* order.

(6) *Flight* Web service confirms cancellation with the *Cancelled* message.

Figure II.7: BTP atomic business transaction

WS-Transaction [Microsoft, BEA and IBM, 2004a] defines a specialization of WS-Coordination, which is an extensible framework for specifying distributed protocols that coordinate the execution of Web services, and that can be used in conjunction with BPEL [Microsoft, BEA and IBM, 2004b]. Like BTP, it offers



(1) *Travel Agent* sends the request messages to the two flight booking Web services, *Air France* and *British Airways* and to the *Hotel* Web service.

(2) Web services return response messages to the *Travel Agent*.

(3) *Travel Agent* selects *Air France* for the flight booking, and therefore sends a *Cancel* message to *British Airways* Web service and a *Prepare* message to the two other Web services.

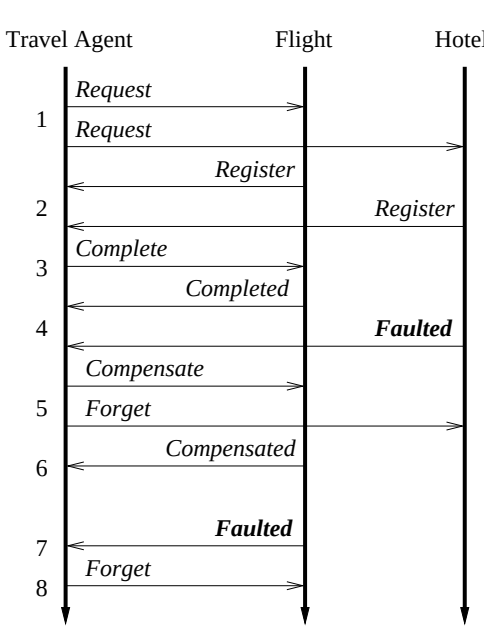
(4) *Air France* and *Hotel* Web services acknowledge with the *Prepared* message and *British Airways* confirms the cancellation with the *Cancelled* message.

(5) *Travel Agent* confirms commits (*Confirm* messages).

(6) Web services acknowledge (*Confirmed* messages).

Figure II.8: BTP cohesive business transaction

two different transaction models: (i) *atomic transactions* (AT) and (ii) *business activity* (BA). An *atomic transaction* adheres to the traditional ACID properties with a two-phase commit protocol. Note that as opposed to the BTP *atomic business transactions*, the isolation property is not relaxed in WS-Transactions, which as we mentioned before, is not suitable for the majority of Web service applications. The *business activity* protocol specifically serves coordinating the execution of open-nested transactions over a set of activities, through a coordinator activity. If there is a need for a coordinated activity to be compensated, the coordinator sends *compensate* messages to all the participants involved in the activity. Then, each participant replies by sending back either a *compensated* or a *faulted* message, depending on whether the required compensation operation was successfully completed or not. However, there is no requirement for an agreement on the outcome, and any participant can leave the coordinated activity in which it is engaged, prior to the termination of peer participants. A WS-Transaction *business activity* example is shown in Figure II.9, with an *Airline* Web service and an *Hotel* Web service.



(1) The *Travel Agent (TA)* initiates the *Business Activity* with the *Flight (F)* and *Hotel (H)* participants (*Request* messages).

(2) *F* and *H* enroll in the transaction (*Register* messages).

(3) *TA* initiate booking on *F* (*Complete* message).

(4) *F* returns *Completed* to confirm commitment, while *H* returns a *Faulted* error message.

(5) For aborting the whole transaction *TA* sends a *Compensate* message to *F* which has already completed the (sub)-transaction and a *Forget* message to *H*.

(6) *F* cancels the booking order and confirms with the *Compensated* message sent back to *TA*.

(7) *F* cannot compensate the booked operation and returns a *Faulted* error message to *TA*.

(8) *TA* sends a *Forget* message to *F*. The flight has been booked and cannot be cancelled.

Figure II.9: WS-Transaction business activity

Although there is not yet a consensus on a standard protocol for managing transactions on the Web, various implementations of these aforementioned protocols are already available[6]. However, these solutions have a restricted applicability when composing autonomous Web services because they require that the composed Web services enrolled in the transaction support one of these protocols. Indeed, Web services should understand the protocol messages they receive and should sent back well-defined protocol messages. Furthermore, using primarily transactions for tolerating faults do not cope with all the specifics of Web services. A major source of penalty lies in the use of backward error recovery in an open system such as the Internet. Backward error recovery is mainly oriented towards tolerating hardware faults but poorly suited to the deployment of cooperation-based mechanisms over autonomous component systems that often require cooperative application-level error recovery among component systems. Moreover, cancellation or compensation does not always work in many real-life situations, which involve documents, goods, money as well as humans (clients,

[6]JOTM transaction manager (http://www.objectweb.org/jotm/) and Cohesions (http://www.choreology.com/) implement the BTP protocol and Oracle BPEL Process Manager (http://www.oracle.com/appserver/bpel_home.html) and Arjuna XML Transaction Service (http://www.arjuna.com/products/arjunaxts/) support the WS-Transaction protocols.

operators, managers, etc.) and which require application-specific error handling. Forward error recovery mechanisms should thus be provided as a complementary mean for achieving dependability.

II.3.3 Forward error recovery for the Web

Exception handling is the main mechanism used to achieve forward error recovery [Goodenough, 1975, Cristian, 1989]. It consists of specifying the behaviour of a software system when faults are detected during its execution. When an error occurs an exception is *raised* causing the normal execution flow of the program to be interrupted and an *exception handler* is executed for recovery. Then, either the process terminates (termination model), or it resumes at the point where the exception had been raised (resumption model). Exceptions and associated handlers are generally specified during the development phase using adequate programming language constructs. Applying exception handling in distributed systems that involve concurrent processes raises several challenges that have led to the design of specialized fault tolerance mechanisms. In particular, there is a need for adequate structuring mechanisms for dealing with exceptions raised in some components of a system while other components execute normally and with concurrently raised exceptions [Campbell and Randell, 1986, Issarny, 1993].

Exception handling mechanisms for realizing forward error recovery are extensively exploited in the specifications of composite Web services in order to handle error occurrences (e.g., BPEL [BEA Systems et al., 2005], BPML [A. Arkin, 2002], WSCI [W3C, 2002a], WS-CDL [W3C, 2004b]). The choreography specification language WS-CDL includes constructs for specifying exceptional behaviour of a choreography: when an error occurs, an exception is propagated to roles defined in the choreography using explicit language constructs and the choreography enters an exceptional state. For each exception, an alternative choreography can be defined, which is then executed for handling the exception. In the orchestration language BPEL, exception handlers (referred to as fault handlers) can be associated to a (possibly nested) process, so that when an error occurs inside a process, its execution terminates, and the corresponding exception handler is executed. However, when a process is defined as a concurrent process and at least one embedded process signals an exception, all the embedded processes are terminated as soon as one signaled exception is caught, and only the handler for this specific exception is executed. Hence, error recovery actually accounts for a single exception and thus cannot ensure recovery of a correct state. The only case where correct state recovery may be ensured is when the effect of all the aborted processes are rolled back to a previous state,

which may not be supported in general, in the context of Web services, as discussed previously. This shortcoming of BPEL actually applies to all XML-based languages for Web services composition that integrate support for specifying concurrent processes and exception handling.

Exception handling not only allows defining application-specific recovery methods for dealing with faults, but can also be used to implement and adapt the backward error recovery method that is used according to the actual context of the exception. In addition, the exception handling mechanisms should take into account the concurrent, distributed and open characteristics of the composition processes by providing strong mechanisms for dealing with concurrently raised exceptions as well as language supports for specifying cooperative handling of exceptions among interacting partners.

III Specifying Recovery Support of Web Services

This chapter introduces a conversation language, called WS-RESC, for the specification of both the standard and exceptional, externally visible, behaviour of Web services, further assisting the development of dependable composite services. In a way similar to existing conversation languages, WS-RESC includes constructs for defining ordering constraints over the operations that a Web service provides. However, WS-RESC further includes constructs for specifying support for concurrency, exceptional behaviour, timing constraints and recovery properties of conversation since these are key behavioral properties in the context of dependability. The language in particular enables the definition of equivalence relationships over conversations with respect to their recovery behaviour, which may be exploited for the design of fault-tolerant composite services. In addition, we provide a formal specification of the language through translation into the π-calculus [Milner, 1999] that makes available a large number of tools for reasoning about Web service properties. In particular, it allows the automated analysis of the correct composition of Web services with respect to the services' behaviour.

III.1 The WS-RESC language

The WS-RESC language (Web Service REcovery Support Conversation) is introduced to specify the conversations supported by a Web service for defining the standard and exceptional observable behaviour of the Web service. Conversations are specified in terms of the Web service's offered operations, which are defined in the related WSDL document of the Web service. Conversations and properties holding over them specified in WS-RESC are to be provided as part of the Web service's provided interface, extending the WSDL description. Indeed, it gives to service requesters and to developers of composite Web services, the

information on how to use the service and the properties related to a particular invocation sequence of operations. However, unlike the WSDL document that includes all provided operations, the WS-RESC description may only expose conversations and related properties that are considered important by the Web service designer. Then, other conversations and properties that are relevant to the actual use of the Web service's operations in a specific interaction with a service requester may be obtained by applying composition rules over conversations specified by the Web service.

III.1.1 Conversation modeling

We use state transition systems similarly to UML activity diagram of WSCL (see Figure II.4, page 25) for modeling conversations supported by a Web service: states represent the Web service operation that is called and transitions give the next available operations. Messages can be put as conditions on transitions, which means that the message must be emitted by the operation that has been invoked for the transition to be enabled. The absence of a condition means that no matter the response of the invoked operation, the transition is enabled. The starting state of a conversation is given with a node named *Start*, and final states are represented by nodes named *End*. In addition, we introduce the *Empty* node, which is not associated to any operation. The *Start*, *End* and *Empty* nodes are the only nodes that are not associated to an operation of the Web service. Therefore, transitions originating from *Start* and *Empty* states can not have conditions, and *End* is final.

Throughout this chapter, we use the flight Web service of the travel agent composite Web service case study to illustrate the language constructs. The conversation expressed in WSCL and illustrated previously in Figure II.4 are quite similar as they express the same conversation. Additional notations will be introduced later for describing extra features of the WS-RESC language. Note that, contrary to the WSCL model, we do not model the input and output messages associated to the operations of the Web service at each node as well as the message exchange pattern, as they are already defined in the associated WSDL document.

The definition of WS-RESC comes along with its formal specification through translation in the π-calculus [Milner, 1999], thus allowing for automated reasoning about behavioral matching of Web services (Section II.2.2). As part of our work, we used simulation tests between processes in the discovery process and for the on the fly verification of invocations (see Chapter V). Such a support

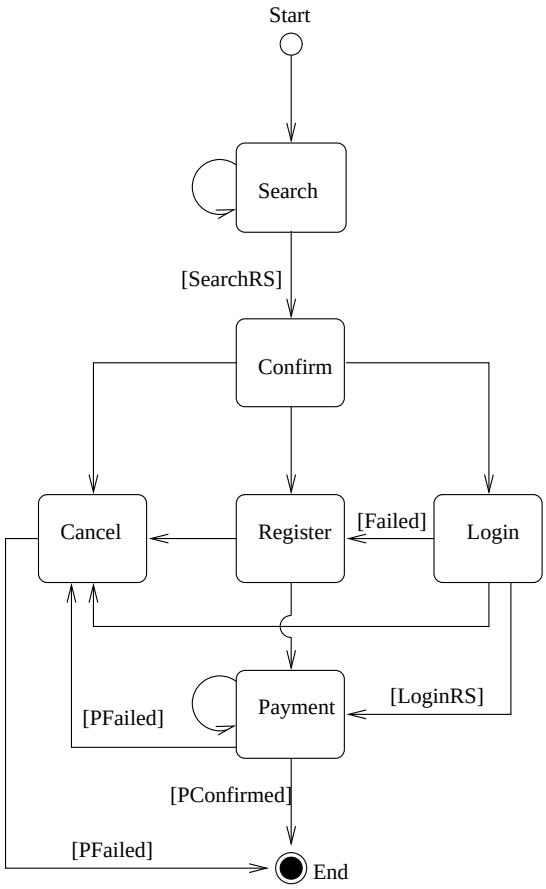


Figure III.1: WS-RESC-based conversation of the flight Web service

is crucial in assisting the development of dependable composite Web services, since it enables enforcing the correct usage of Web services in the composition process.

III.1.2 WS-RESC language constructs

A WS-RESC XML document describes a conversation modeled as a state transition system. We describe each construct of the language using a non-formal XML notation, together with the transition system that it represents and its π-calculus formal definition. The complete XML Schema [W3C, 2004c] definition of the language is further given in Appendix A.

XML notations

The XML notation used here gives XML elements and their attributes as they should appear in a WS-RESC document, and XML Schema data types are used instead of the values of the attributes. The symbol "?" is used after an attribute or an element to denote that it is optional, the symbol "*" to denote that it can appear 0 or more times, and the symbol "+" denotes that it must appear at least one time and can be repeated indefinitely. Default values of attributes are given with the ":" symbol. Namespaces used are *"this"* that refers to the document being specified and *"ws"* that refers to the associated WSDL document of the Web service for which the conversation support is specified.

π-calculus notations

We use the following notation to denote π-processes, with Exp denoting Boolean expressions:

$P, Q ::=$		Processes
	$P \mid Q$	Parallel
	$P + Q$	Choice
	$!P$	Replication
	$if\ Exp\ then\ P\ else\ Q$	Conditional
	$v(x)$	Input message
	$\bar{v}(x)$	Output message
	$\emptyset$	Null process

The input process $v(x).P$ is ready to input from channel v, then to run P with the formal parameter x replaced by the actual message, while the output process $\bar{v}(y).P$ is ready to output message y on channel v, then to run P. The *reduction* relation, noted $\rightarrow$, is further defined over processes, with $P \rightarrow P_1$ expressing that P can evolve to process P_1 as a result of an action within P. For instance, we have: $((\bar{v}(x).P + P') \mid (v(y).Q + Q')) \rightarrow P \mid Q\{^x/_y\}$, with $Q\{^x/_y\}$ meaning that x replaces y in Q. In the following, we also use a shorthand notation for input and output messages, denoting the channel and parameter with message names as in $P ::= in.(\overline{out}.Q)$.

III.1.2.1 Sequencing

The main elements of a conversation are the states and transitions, which give the ordering requirements over invocations of the operations of the Web service.

States are associated to an operation of the Web service and are declared using the *state* element with three attributes.

```
<state name=NCName
       operation=QName
       correlate=QName ? />
```

- *name* of type *NCName* (an XML name, without the namespace part), gives a name to the state. It is used to reference the state, allowing thus to reuse the state definition in different transition rules.
- *operation* of type *QName* (an XML name prefixed with a namespace) references an operation of the Web service.
- *correlate* of type *QName* is optional and is used for identifying different sessions.

Transition from a source state to a destination state occurs when the operation associated to the source state is executed and the execution terminated. Destination states then denote the operations that can be subsequently called. Additionally, we may have conditions on transitions, represented as labels on the transitions. The condition can be set on output or fault messages of the operation that is executed (the operation referenced in the source state of the transition). When set, it states that the transition is valid only if the operation being executed returns the given message, specified by its name.

A transition is specified using the *transition* XML element, which embeds: the *source* element that gives the source state, and the *destination* element that gives the target state:

```
<transition
      name=NCName >
   <source
      state=QName
      condition=X-Path-expression ? />
   <destination
```

```
        state=QName
        minOccurs=nonNegativeInteger : 1
        maxOccurs=(nonNegativeInteger | unbounded) : 1
    />
</transition>
```

The optional *condition* attribute of the *source* element is defined for transitions that depend on some output or fault messages. The condition is expressed as an XPath[W3C, 1999] expression and may thus be a Boolean expression composed of several messages. In addition, we use the attributes *minOccurs* and *maxOccurs* for the *destination* element to specify how many times the transition should be repeated, used in general for transitions on the same state.

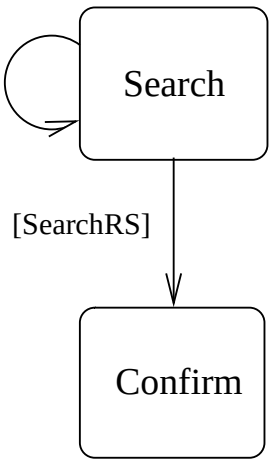
A state named A and its related transition into a state B directly translates into a π process $A_\pi ::= in.((\overline{out}.B_\pi))$, with in being the input message for the operation associated with A and , $\overline{out}.B_\pi$ modeling the transition labeled with message abstracted by out, and B_π denoting the process associated with the destination state.

As an illustration, Figure III.2 defines the transitions from the *Search* operation of the flight reservation Web service. The operation *Search* can be called as many times as necessary, followed by a confirmation by invoking the operation *Confirm*, under the condition that the last search returns the *SearchRS* message.

III.1.2.2 Activities

A conversation supported by a Web service is defined as an activity. A Web service may support more than one conversation, which should be defined using distinct activities in a single WS-RESC document. The top-most XML element in a WS-RESC document is an *activity*, which defines a set of transitions, defining a connected graph. Furthermore, activities may be composed in order to define broader activities.

```
<RESC>
    <activity name=NCName ref=QName ?> *
        <transition ...> *
        Exceptional statements (defined later)
        Properties (defined later)
    </activity>
</RESC>
```

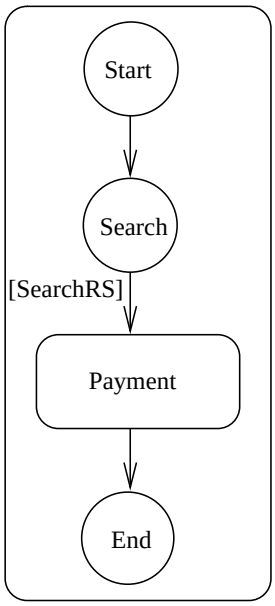



```
<state name=''Search''
  operation=''ws:Search''/>
<state name=''Confirm''
  operation=''ws:Confirm''/>

<transition name=''t1''>
  <source state=''this:Search''/>
  <destination state=''this:Search''
    minOccurs=0 maxOccurs=unbounded
  />
</transition>

<transition name=''t2'' >
  <source state=''this:Search''
    condition=''ws:SearchRS''/>
  <destination state=''this:Confirm''/>
</transition>
```

Figure III.2: Ordering of operations

Activities are named and contain at least a transition from a *Start* state and zero or more transitions to *End* states. Furthermore, we allow reuse of pre-defined activities with the *ref* attribute by referencing its name. All other transitions defined in the activity may reference a Web service operation or another activity. The contained activity is referred as a nested activity and the containing one as the main activity, which has its own *Start* and *End* states. A nested activity has to terminate on an *End* state to allow the continuation of the main activity. A nested activity is viewed as an isolated execution, considered as a single state in the diagram. Isolation is further enforced by disallowing states to be shared between activities. Activities directly translate into π-calculus processes, according to rules associated with embedded constructs.

As an illustration, Figure III.3 depicts an activity with a nested activity *Payment*, which defines the sub-conversation for the authentication of the service requester and the payment. Activities are declared using the *activity* element and nesting is specified through transition with a destination element of type *activity*, leading to an implicit transition on the *Start* state of the nested activity. The nested activity may then continue until an *End* state is reached. Only then, the containing activity may resume, from a transition that has the nested activity as a source destination. Conditions on this transition may be set on the output or fault messages of the state preceding the *End* state in the nested activity.




```
<activity name=''Flight''>
  <transition name=''t''>
    <source state=''this:Search''
      condition=''ws:SearchRS''/>
    <destination state=''this:Payment''/>
  </transition>
  ...
</activity>

<activity name=''Payment''>
  ...
</activity>
```

Figure III.3: Activities and composition

III.1.2.3 Choice

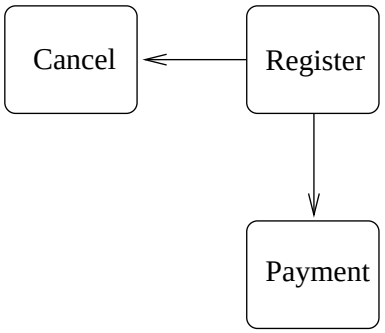
The choice operator allows defining non-deterministic exclusive choices among a set of operations that can be called based on the service requester selection. Once an operation is executed, other options are no longer valid.

Using XML, choices are specified by declaring transitions with multiple destinations:

```
<transition name=NCName>
   <source state=QName
           condition=XPath-expression ? />
   <destination state=QName /> +
</transition>
```

The choice construct directly translates into π processes combined with the choice $(+)$ operator, as in $A_\pi ::= in.((\overline{out_1}.B_\pi^1) + ... + (\overline{out_n}.B_\pi^n)$.

For example, in the conversation described in Figure III.4, it is specified that after one execution of the *Register* operation, the service requester is allowed to call either the *Cancel* operation or the *Payment* operation.




```
<transition name=''t''>
  <source state=''this:Register''/>
  <destination state=''this:Cancel''/>
  <destination state=''this:Payment''/>
</transition>
```

Figure III.4: Choice

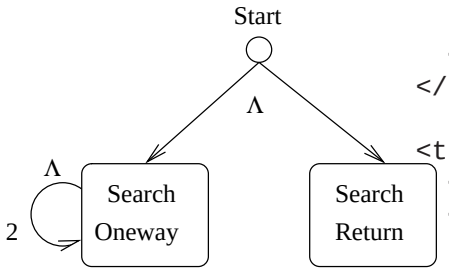
III.1.2.4 Concurrency

The concurrency construct serves specifying operations or activities that are allowed to be executed concurrently by a service requester. The corresponding XML declaration is given using the *concurrent* element, as a replacement of the *destination* element in the *transition* element. All destination states referring concurrent operations and concurrent activities are declared as a child element of the *concurrent* element. Concurrency can also be defined for a single state, i.e., for an operation or activity that may be called several times in parallel. The *maxOccurs* attribute on a *destination* element gives the maximum number of concurrent calls that is allowed for the referenced operation or activity.

```
<transition name=NCName >
   <source state=NCName
           condition=XPath-expression ? />
   <concurrent>
      <destination state=QName
         maxOccurs=(nonNegativeInteger|unbounded):1>+
   </concurrent>
</transition>
```

The concurrency construct directly translates in the π-calculus using the parallel (|) operator.

Figure III.5 depicts a conversation where the transition from the *Start* state leads to two concurrent sub-activities: *SearchOneWay* and *SearchReturn*. Concurrency is specified using the $\wedge$ symbol on the diagram. Furthermore, the optional number n on the transition states the maximum number of parallel executions of the activity that is allowed (or * for specifying unlimited concurrent calls). In particular, the service requester can call concurrently the *SearchOneWay* operation




```
<transition name=''concurrency''>
  <source state=''this:Start''/>
  <concurrent>
    <destination state=''this:SearchOneWay''/>
    <destination state=''this:SearchFlight''/>
  </concurrent>
</transition>

<transition>
  <source state=''this:SearchOneWay''/>
  <concurrent>
    <destination state=''this:SearchOneWay''
      maxOccurs=2/>
  </concurrent>
</transition>
```

Figure III.5: Concurrency

twice.

III.1.2.5 Identifying sessions

The concurrency construct introduced above set the rules for concurrent interactions between a Web service and a service requester. When several service requesters access concurrently a Web service, concurrency control is internal to the service provider side. For example, concurrency control can be managed either by the Web service application or by the application server, which can create as many Web service instances as requests or as service requester and redirect incoming messages to the related instance. Each interaction with a different Web service requester can thus be considered as a separate conversation. Nonetheless, a single service requester can also interact with a Web service in different sessions. Different conversations should then apply for each of these sessions, as if they were undertaken with different service requesters. The service requester must however know if the Web service can identify different sessions with respect to conversations. In this case the sessions can be executed concurrently as each of them will be matched to a different conversation instance. Otherwise, concurrency control should be made at the service requester side, e.g., by serializing concurrent conversation execution instances.

Since there is not a standard way to manage sessions in Web services, keeping

track of different Web service interaction instances is usually managed by applications. For example, a Web service implementation may use cookies stored at the client side for handling sessions, or may require a session identifier to be associated with interactions. Such information may be used to identify the client, in the case where some operations should be invoked within the same session by the same service requester, as well as to identify a specific Web service instance on the service provider-side.

Web services should thus make visible their support for concurrent sessions at their interface. This is done in WS-RESC by associating states and activities of a conversation with abstract correlation values, which can be tracked by the service requester to identify different sessions. The absence of correlation information means that the Web service does not support concurrent conversations (for activities not defined as being concurrent explicitly). When a state or activity has the same correlation value as another, this means that both states are part of the same session. We define correlations with the element *correlation*. We get the following definition:

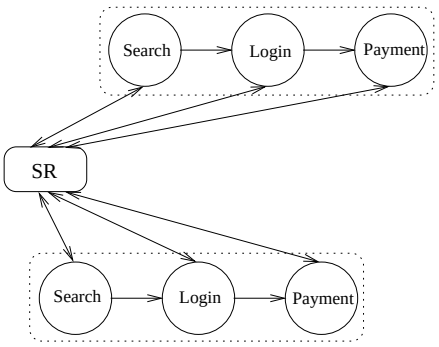
```
<correlation name=NCName />
```

Then, states sharing the same correlation values are identified through the attribute *correlate* referencing this correlation element. For activities, the correlation applies automatically to all embedded states and activities:

```
<correlation name=NCName />
<state name=NCName
       operation=QName
       correlate=QName />
<activity name=NCName
       correlate=QName />
```

Correlations are defined in corresponding π processes using parameters. Specifically, $(\nu i)A_\pi(i)$ is the process A_π with the newly defined correlation value i. The correlation value can then be shared with processes within the same session by passing the value in a way similar to input messages.

In Figure III.6, the service-requester *SR* initiates two concurrent interactions, each mapped to the conversation *(Search.Login.Payment)*. The conversation is not a concurrent conversation as it would be defined with a *concurrent* construct because each interaction belongs to two distinct sessions. However, the service requester can initiate two concurrent conversations with the Web service because a correlation identifier has been defined for each state of the conversation for identifying which operation call belongs to which conversation instance. Note




```
<correlation name=''SessionID''/>
<state name=''Search''
  operation=''ws:Search''
  correlate=''SessionID''/>
<state name=''Login''
  operation=''ws:Login''
  correlate=''SessionID''/>
<state name=''Pay''
  operation=''ws:Pay''
  correlate=''SessionID''/>
```

Figure III.6: Concurrent sessions

that in this specific example, the correlation information could also have been set once, at the level of the main activity.

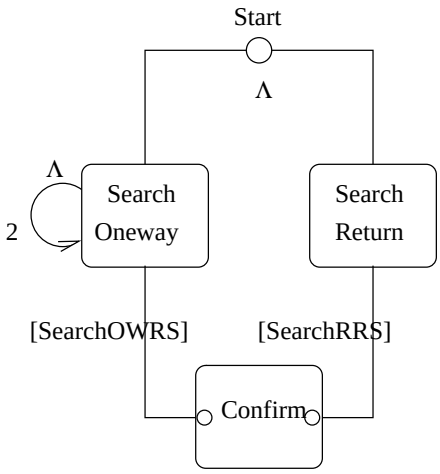
III.1.2.6 Synchronization

Synchronization of concurrent activities is specified using join conditions. A join condition is expressed as a Boolean expression on output messages of operations that specifies the condition under which the execution of the conversation is allowed to continue.

Various join conditions can be specified, ranging from the synchronization of all the parallel activities, termination of a subset of the concurrent activities, and no condition at all, meaning that these activities are not required to terminate before the execution of activities after the join. In general, the join condition is specified as a Boolean expression on the output messages of the last operation of each activity that is joined. We get the following definition:

```
<transition name=NCName>
   <source state=QName
           condition=XPath-expression ? />+
   <destination state=QName
           condition=XPath-expression ? />
</transition>
```

Formally, this translates into a conditional π process that is sequentially composed with the concurrent activity, and whose condition, if any, is expressed as a Boolean expression over related output events.




```
<transition name=''confirmation''>
  <source state=''this:SearchOneWay''
    condition=''ws:SearchOWRS''/>
  <source state=''SearchReturn''
    condition=''ws:SearchRRS''/>
  <destination state=''this:Confirm''
    condition=
    ''ws:SearchOWRS AND ws:SearchRRS''/>
</transition>
```

Figure III.7: Synchronization of concurrent activities

Execution paths of two concurrent activities that have been split are merged into a single state, as shown in the activity depicted in Figure III.7. In this example, the concurrent activities are both required to terminate before invoking the *Confirm* operation. The corresponding transition specifies the states of all the concurrent activities that are to be joined as source states, and the join condition as an Xpath[W3C, 1999] expression on the destination state.

III.1.2.7 Timing constraints

In Web service interactions, both the service requester and the service provider can have timing requirements over their interactions. A service requester can assume that a Web service has failed if an operation call does not returns an output or a fault message within a pre-determined amount of time. On the other hand, service providers may also set timeouts on input messages received from a single service requester. WS-RESC enables specifying such timing constraints for Web services over transitions.

Since there is not a standard way to model timers in the π-calculus, we can abstract from time by using a specific process that is run when the timer timeouts and that can be prefixed by an output event relating to a timeout fault message, if any. A timer set on an operation *Login* that returns a *TimeOut* fault message may then be specified as:

$$Login_\pi.(\overline{OK}.Search_\pi + \overline{TimeOut}.Login_\pi)$$

An alternative would be to model the time directly in the calculus, as presented in [Berger and Honda, 2003], where the authors extend the π-calculus with a timer denoted by:

$$timer^t(\bar{x}(v).P, Q)$$

where t is a positive integer representing time steps, Q the process that is run when the timer timeouts and $\bar{x}(v).P$ is the continuation process. The latter modeling has the advantage of making time explicit and thus allows reasoning about timing properties of processes. However, we undertake the former approach for modeling timeout since it is directly supported in the π-calculus.

In WS-RESC, a timeout is associated with a transition using the *timeout* element, defined as follow:

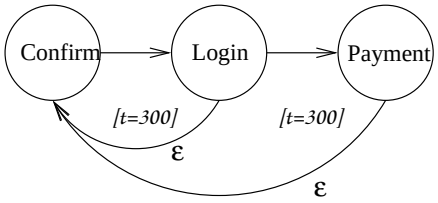
```
<timeout timer = Duration
         onInput = Boolean : false
         state = QName ?
         exception = QName ?
/>
```

The embedded *onInput* attribute is set to *true* if the timeout is computed from upon receipt of an input message for the operation referenced in the source state, or set to *false* (default) if the timeout is computed from upon emission of the output message by the Web service. The *state* attribute further specifies the destination state of the process upon timeout occurrence. If no state is specified, the *exception* attribute is used to reference an exception that will be raised on timeout.

For example, the flight Web service can require a delay of 15 minutes between the confirmation and the login, and between the registration and the payment. On timeout, the service requester should re-search or re-login, as illustrated in Figure III.8.

III.1.3 Exceptional behaviour

The exceptional behavior of a Web service is described by giving an alternative conversation that should be followed by the service requester when it detects an




```
<transition>
  <source state=''this:Confirm''/>
  <timeout duration=''300''
    onInput=''false''
    state=''this:Confirm''
  <destination state=''this:Login''/>
</transition>

<transition>
  <source state=''this:Login''/>
  <timeout duration=''300''
    onInput=''false''
    state=''this:Login''
  <destination state=''this:Payment''/>
</transition>
```

Figure III.8: An activity with timers

exceptional behaviour. The occurrence of an exception is identified by the service requester, according to the *exception triggering conditions*. These conditions are specified by the service provider for each activity supported by the Web service, which might raise an exception. An exception is considered to be *raised* if all triggering conditions defined for this exception are met. Then, the activity is considered as terminated, and interactions with the service requester and the Web service should follow the corresponding exceptional conversation (the *exception handler activity*). An exception handler activity always terminates with a transition on the *End* state of the activity for which it is defined, causing it to terminate also. If the activity is a sub-activity, then the conversation continues as if the activity had terminated normally.

If a corresponding exception handler cannot be found for an exception, the activity, which has raised the exception, terminates and the exception is propagated to the containing activity –if any. The propagation is repeated until a corresponding handler is found. If no handler can be found, the whole conversation is considered to be terminated exceptionally.

Exception handler activities are specified in the same way as other activities, and can thus raise exceptions for which other handler activities may be defined. However, exceptions are never propagated as is across different handlers. But, if a handler terminates exceptionally, it may trigger an exception in a parent activity, e.g., a parent exception handler, where it should be defined. This is called *exception signaling*.

Exception triggering conditions comprise:

(1) *Error messages*, sent by the Web service as either a WSDL fault message or as an output message.

(2) *Service failures*, detected by the underlying communication protocol of the service requester (e.g., HTTP errors or SOAP fault messages).

(3) *Timeouts*, occurred when a service requester does not meet timing conditions on operations invocations.

In WS-RESC exceptions are declared by embedding *exception* elements within *activity* elements:

```
<exception name=NCName
           condition=(XPath-expression | any)/>
```

The *exception* element has two attributes:

(1) *name* for naming the exception. The name must be uniquely defined in its context. It is used by the handler activity to reference the exception. The handler activity is specified separately, in the same WS-RESC document.

(2) *condition* for setting the condition that must be satisfied for raising the exception.

Conditions are expressed as an XPath expressions over other exceptions or messages. Service failures are specified using XML QNames referencing the specific failure. In particular, having an exception name in a condition expression enables defining propagation of exceptions across activities and handlers. In addition, the keyword *failure* is used to reference any unhandled exception that might be propagated or signalled in the context. Exception handler activities are further defined using the *handler* elements, refering the name of the associated exception and a reference to an activity defining the handler, which can further define other exceptions. When the handler activity terminates normally, i.e., by reaching the *End* state without throwing any other exception, the parent activity or handler that was interrupted by the exception being handled terminates by reaching the *End* state.

An example conversation is for illustrating exception handling and exception propagation is presented in Figure III.9. The main activity named *A* is composed of two sub-activities *A1* and *A2* that should be executed in sequence. An

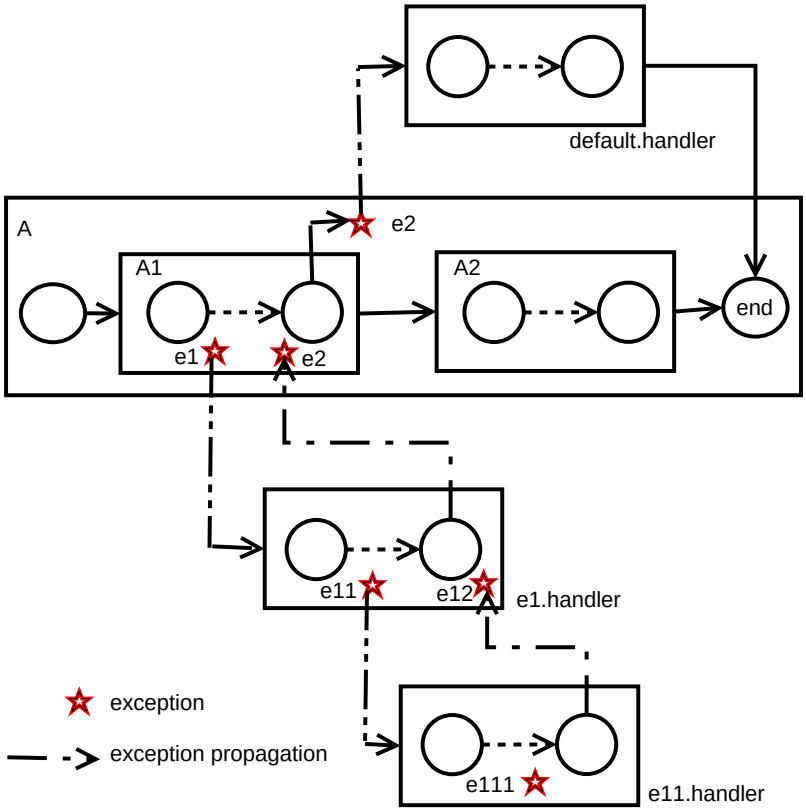


Figure III.9: Exception handling

exception named *e1* is defined for the activity *A2* and leads the handler activity *e1.handler* to be invoked, interrupting the normal execution of activity *A1*. Similarly, the *e1.handler* activity throws the *e11* exception, leading the service requester to invoke operations of the *e11.handler* activity. A third exception *e111* is thrown in *e11.handler*, but it has no associated exception handler. The exception *e111* is then propagated recursively up to the main activity *A*. For propagating the exception across exception handlers, a new exception must be raised (exception signalling). *e111* becomes thus *e12* when signalled to *e1.handler* and *e12* becomes *e2* in *A1*. *e2* is propagated as is to its parent activity *A*. A default exception handler *default.handler* matches this exception, and can be executed. When the service requester completes all the conversation described by the *default.handler* activity, the main activity terminates with a transition to its *End* state. Note that the *A2* activity have never been executed because of the exception *e2* in the main activity, which terminated it. If the first exception was handled normally and the *e1.handler* activity had terminated, then the execution would have been continued at the *Start* state of the *A2* activity.

The WS-RESC description of activities described in Figure III.9 is given as follows:

```
<activity name=''A''>
   <exception name=''default'' condition=''failure'' />
   <transition>
      <source state=''t:A1'' .../>
      <destination state=''t:A2'' .../>
   </transition>
   ...
   <handler exception=''t:default'' activity=''t:defaultHandler''/>
</activity>

<activity name=''A1''>
   <exception name=''e1'' condition=''...''/>
   <exception name=''e2'' condition=''t:e12''/>
   ...
   <handler exception=''t:e1'' activity=''t:e1Handler''/>
</activity>

<activity name=''e1Handler''>
   <exception name=''e11'' condition=''...'' />
   <exception name=''e12'' condition=''t:e111''/>
   ...
   <handler exception=''t:e11'' activity=''e11Handler''/>
</activity>

<activity name=''e11Handler''>
     <exception name=''e111'' condition=''...''/>
  ...
</activity>
```

III.2 Recovery-related properties of conversations

In this section, we present how conversations, in addition to specify how to use the service in terms of dependencies between operations and time constraints, may be used to specify recovery properties of Web services. In a first step, we detail how base recovery properties may be expressed using the notion of state equivalence. However, the internal states of Web services are hidden to requesters and do not make part of the conversation definitions, which exhibit only the potential behaviour of Web services. Then, we discuss how state equivalence that is based on the knowledge of the systems' internal states may be introduced in the modeling of systems that exhibit only their observable behavior, leading us to define a specific equivalence relation over conversations.

III.2.1 Equivalence relation for expressing recovery-related properties

Most of the recovery mechanisms implemented in composite Web services such as open-nested transactions, or retry-based recovery actions, require calling operations or sequence of operations on composed Web services that have recovery-related properties such as atomicity, compensability, retry-ability and commutativity. These properties can be characterized in terms of state equivalence (also referred to as final state equivalence) relationships over values of the individual services' states.

A retry-able activity can be repeated several times until it succeeds. Retry-ability often involves using idempotent activities, i.e., activities that, when executed several times, give the same result. In other words, an activity is retry-able if the internal states reached after one or more sequential executions of the activity are equivalent.

Atomicity is a base recovery property for running transactions stating that an activity either successfully executes until completion, or aborts by exceptionally terminating in the same state as the one that held before its execution. Similarly, open-nested transactions for long running activities are realized by calling operations that compensates previous ones. Compensation-based activities can be defined using the notion of state equivalence, specifying that the successful compensation of an operation, or of a set of operations, brings the system to a state that is equivalent to its initial state.

The above definition assumes that the compensation action is performed immediately after the action that is to be compensated: the initial state of the system before the execution of the compensation activity is equivalent to the system state after the execution of the compensated activity. If other actions are performed in between, as it is possible with long-running transactions, the internal state of the system may change, and the compensation action can no longer ensure the state back recovery based on this definition. In [Korth et al., 1990], the authors formally define compensating transactions based on the equality of histories. The definition uses the notion of the commutativity of sequence of operations. Then, different types of compensations are defined based on this notion. In particular, if it is stated that two actions P and Q commute, and R is the compensation operation of P, , then executing the sequence $(P.Q.R)$ has the same effect than executing the sequence $(Q.P.R)$, where Q is executed and P is compensated. Commutativity of operations can be expressed using the state equivalence relation stating that two operations commute if whatever is

the execution sequence of the operations, the internal state of the system would be the same after the execution of the two operations.

III.2.2 An equivalence relation over conversations

According to [Gaudel et al., 2003], there are two different approaches for defining the internal state of a system:

(1) Forward-looking style with which the internal state of a system at a given instant is a notional attribute of the system that is sufficient to determine the system's potential behavior;

(2) Backward-looking style with which the internal state of a system is the total information explicitly stored (in state variables) by the system up to the given instant.

The state equivalence relation that is used in the previous section for specifying recovery-related properties relies on the equivalence of internal states of systems, as given in the second definition. However, conversation languages define the observable behaviour of Web services where the internal state remains hidden. Indeed, what is typically described in conversation languages follows the former definition, i.e., the potential behavior of systems. The system is specifically viewed as a process and represented, in general, with labeled transition systems. Using these modeling approaches, the notion of equivalence (referred to as observational equivalence) is expressed in terms of the system's external behavior and verified using bisimulations of processes (e.g., see [Sangiorgi and Walker, 2001] for an exhaustive list of different bisimulations for the π-calculus[Milner, 1999]).

To be able to express the properties defined in the previous section, we need a new equivalence relationship over conversations, to specify the equivalence of the system's internal states after the execution of operations of a given conversation, without making explicit their values, in a way similar to the work of [Black et al., 2003].

We further use the term *conversation execution* to denote such an execution process, i.e., the execution of a conversation is defined by a process execution path that *matches* the given conversation and reaches any final state.

Let P a conversation, σ the internal state of the system as perceived by the service requester and α a conversation supported by P, which when executed reduces P to the null process. The post execution internal state of the system evolves to σ':

$$(P, \sigma) \xrightarrow{\alpha} (\emptyset, \sigma')$$

We introduce the equivalence relation as a binary relation, noted $\tilde{\sim}$, between two conversations:

Definition III.1 *For two alternative conversations P and Q defined for a Web service, if $P \tilde{\sim} Q$ holds, then the internal state of the Web service after an execution of P would be equivalent to that reached after an execution of Q, if the initial internal states for both alternative executions are equivalent.*

Formally,

$$\begin{aligned} P \tilde{\sim} Q \equiv \quad & \forall \alpha, (P, \sigma_P) \xrightarrow{\alpha} (\emptyset, \sigma'_P), \\ & \forall \beta, (Q, \sigma_Q) \xrightarrow{\beta} (\emptyset, \sigma'_Q), \\ & \sigma_P = \sigma_Q \implies \sigma'_P = \sigma'_Q \end{aligned} \tag{III.1}$$

Note that the equivalence relation defined above specifies only equivalence over internal states, not over their observable behaviours. Thus, conversations over which the equivalence relationship holds are not necessarily structurally congruent nor observationally equivalent. Our equivalence relation satisfies the following properties:

(1) Reflexivity: $P \tilde{\sim} P$

(2) Symmetry: $(P \tilde{\sim} Q) \Rightarrow (Q \tilde{\sim} P)$

(3) Transitivity: $((P \tilde{\sim} Q) \wedge (Q \tilde{\sim} R)) \Rightarrow (P \tilde{\sim} R)$

Proofs are trivial given the properties of the equality relation over states:

(1) $\forall \alpha, (P, \sigma) \xrightarrow{\alpha} (\emptyset, \sigma'), \sigma' = \sigma' \implies P \bar{\sim} P$

(2) $\forall \alpha, (P, \sigma_p) \xrightarrow{\alpha} (\emptyset, \sigma'_p)$ and $\forall \beta, (Q, \sigma_q) \xrightarrow{\beta} (\emptyset, \sigma'_q)$.
$P \bar{\sim} Q \implies \sigma'_p = \sigma'_q. \implies \sigma'_q = \sigma'_p \implies P \bar{\sim} Q.$

(3) $\forall \alpha, (P, \sigma_p) \xrightarrow{\alpha} (\emptyset, \sigma'_p)$ and $\forall \beta, (Q, \sigma_q) \xrightarrow{\beta} (\emptyset, \sigma'_q)$ and $\forall \gamma (R, \sigma_r) \xrightarrow{\gamma} (\emptyset, \sigma'_r)$,
$P \bar{\sim} Q \implies \sigma'_p = \sigma'_q$ and $Q \bar{\sim} R \implies \sigma'_q = \sigma'_r \implies P \bar{\sim} R$

These properties can then be used to establish equivalence relations over larger conversations of a single Web service by composing smaller ones defined by the developer of the Web service.

III.3 Expressing recovery-related properties

We provide two different methods for specifying the recovery properties of a conversation. The first method is to declare the recovery property similarly to existing solutions on the area, by annotating conversations with pre-defined properties ([Benatallah et al., 2004, Jimenez-Peris et al., 2003]). The second method does not specify the recovery property directly, but uses the equivalence relationship introduced previously for specifying equivalence relationships between different conversations supported by the Web service. The recovery property is then derived from the specification by the service requester, which can also compose different conversations of a single Web service to establish new equivalence relations that were not explicitly declared by the service provider.

III.3.1 Expressing recovery properties using meta-data

A property is associated to an activity using the *property* child element. Properties are referenced by their QNames. A service requester can then select a Web service based on the property that it exhibits over its conversations.

```
<activity>
    <property value=QName/>*
    ...
</activity>
```

For example, if the properties defining a transaction are defined in an XML document referenced using the *trans* XML namespace, for a conversation satisfying the all the ACID properties we get:

```
<activity>
   <property value=''trans:atomicity''/>*
   <property value=''trans:consistency''/>*
   <property value=''trans:isolation''/>*
   <property value=''trans:durability''/>*
    ...
</activity>
```

This method is introduced as a complementary approach to define properties and in particular, for those that can not be specified using the equivalence relation. However, as the semantics of the properties are not explicitly specified, we can not derive properties of conversations obtained by composing existing ones.

III.3.2 Expressing equivalence relations in WS-RESC

In WS-RESC, we introduce the *equivalence* element to specify equivalence between activities (normal and exception handler activities). We get the following XML notation for declaring that an execution matching a conversation given by the activity A is equivalent to an execution matching a conversation given by an activity B:

```
<equivalence>
   <activity ref=''this:A''/>
   <activity ref=''this:B''/>
</equivalence>
```

We recall that the equivalence relation serves specifying equivalence of internal states and not of behavior. Thus, our definition does not map to any of the equivalence relationships defined over π processes.

In the following section, the equivalence relation is used to express recovery properties of conversations.

III.3.2.1 Expressing alternative conversations

A first usage pattern of equivalence relations is to declare alternative execution paths, which can have express different behaviours in term of the sequence of operations, but give the same results in terms of the internal state change of the Web service. Given two activities A and B, if the equivalence relationship

$A \sim B$ holds, then the service requester can substitute an activity by the other one. Note that the input and output messages of different operations included in each activity may be different, and it is up to the service requester to decide which activity should be effectively executed and to set correct values to message parameters.

```
<equivalence>
   <activity ref=''this:A''/>
   <activity ref=''this:B''/>
</equivalence>
```

Alternative activities can indeed be used for achieving dependability by choosing an alternative execution if the execution of an activity fails or if it can not be executed by the service requester, for example if some messages can not be constructed to call an operation in one of the activities.

III.3.2.2 Expressing retry-ability

As mentioned in Section III.2.1, a retry-able activity often involves using idempotent activities. An idempotent activity A can be expressed by declaring an equivalence relationship between A and an activity defined using the activity A with a transition on itself:

```
<equivalence>
    <activity ref=''this:A''/>
    <activity>
        <transition>
            <source state=''this:A''/>
            <destination state=''this:A''/>
        </transition>
    </activity>
</equivalence>
```

III.3.2.3 Expressing rollback

Specifying abort semantics of atomic conversations or compensation activities is done by expressing the restoration of the internal state to the initial state. In WS-RESC we set an equivalence relation between the rollback activity, which is executed after the activity to cancel, and the activity executed before or the initial *Start* state.

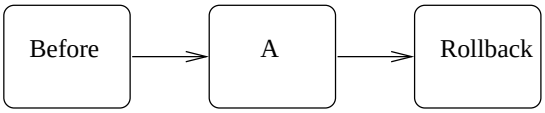


Figure III.10: Rollback activity

Given the conversation illustrated in Figure III.10 defined with transitions over activities *A*, *Rollback* and *Before*:

```
<transition>
   <source state=''this:Before''/>
   <destination state=''this:A''>
</transition>

<transition>
   <source state=''this:A''/>
   <destination state=''this:Rollback''>
</transition>
```

Cancellation of the effects of activity *A* on the internal state of the Web service is given by the relation:

```
<equivalence>
   <activity ref=''this:Rollback''/>
   <activity ref=''this:Before''/>
</equivalence>
```

The equivalence can be established also with only the *Start* state, indicating that all actions have been cancelled:

```
<equivalence>
   <activity ref=''this:Rollback''/>
   <state ref=''this:Start''/>
</equivalence>
```

III.3.2.4 Expressing commutativity

Commutativity of two activities A and B can be expressed with an equivalence relation between an activity embedding A and B, with a transition from A to B and an activity embedding A and B, with a transition from B to A:

```
<equivalence>
   <activity>
        <transition>
            <source state=''this:A''/>
            <destination state=''this:B''/>
        </transition>
   </activity>
   <activity>
        <transition>
            <source state=''this:B''/>
            <destination state=''this:A''/>
        </transition>
   </activity>
</equivalence>
```

III.4 Case study

This section illustrates the usage of the WS-RESC language for specifying equivalence relations over conversations, from which the recovery behaviour of the Web service can be derived. We extend for this the *Flight Web Service* introduced in Section III.1.1.

III.4.1 Retry-ability

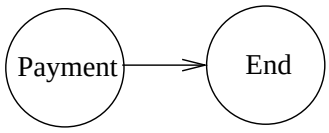


Figure III.11: Retry-able conversation

Consider the part of the conversation depicted in Figure III.11 representing the payment of the booked flight. It may be useful for the service requester to know that the internal state of the Web service (which is hidden) is exactly the same if the payment operation is to be invoked multiple times sequentially, i.e., that the operation is *retry-able*. The service requester may use this information either to verify if the flight Web service supports retry-ability in case of failure (e.g.,

timeout when performing an operation), or to implement an application-specific forward error recovery based on the retry technique.

The equivalence relation for specifying the retry-ability property of the *Payment* activity is given by:

```
<equivalence>
    <activity ref=''this:Payment''/>
    <activity>
        <transition>
            <source state=''this:Payment''/>
            <destination state=''this:Payment''/>
        </transition>
    </activity>
</equivalence>
```

III.4.2 Atomicity

An activity is said to be atomic if it terminates either successfully by committing performed operations, or without completing its task by aborting and restoring its state back, as illustrated in the example in Figure III.12. The conversation starts with the invocation of the *Login* activity, which comprises all the *Search, Register* and *Login* operations. The conversation then continues by multiple invocations of the *Payment* operation. If any of these *Payment* operation calls returns a fault message, the whole activity is aborted on the server side. The service requester can no longer proceed for a *Payment* and can only call the *Cancel* operation to confirm the abortion. Otherwise, the client can terminate anytime or call the *Cancel* operation to cancel the effects of previous operations. Note that in the former situation, the service requester calls the *Cancel* operation to confirm (i.e., acknowledge) the abortion that is automatically done by the Web service, and that in the latter, the choice of either aborting or validating is left to the the requester.

Atomicity of the sequence *Login, Payment* and *Cancel*, which is an activity supported by the flight Web service as it can be simulated (in the π-calculus sense) by the conversation modeled in Figure III.12, is expressed with an equivalence relation with the conversation *Login*. The *Cancel* operation has thus undone the effects on the server side of the *Payment* operation, as the state as it is perceived by the service requester (in particular, its bank account) is equivalent to the state that would be reached if it only executed the *Login* operation.

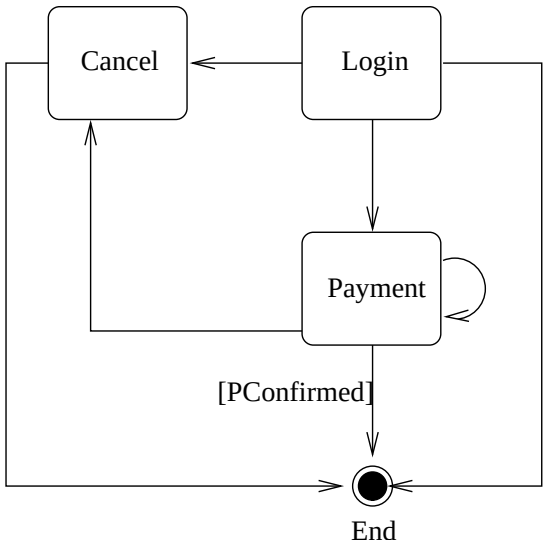


Figure III.12: Atomicity

The specification in WS-RESC is as follows:

```
<equivalence>
   <activity ref=''Login''/>
   <activity>
      <transition>
          <source activity=''this:Login''/>
          <destination state=''this:Payment''/>
      </transition>
      <transition>
          <source state=''this:Payment''/>
          <destination state=''Cancel''/>
      </transition>
      <transition>
          <source state=''this:Cancel''/>
          <destination state=''End''/>
      </transition>
   </activity>
</equivalence>
```

Consider now the conversation that a service requester engages, specified as a successive invocations of the *Payment* operation, followed with the call of the *Cancel* operation. This is a valid execution, since it can be simulated by the conversation supported by the flight reservation Web service and modeled in Figure III.12. However, there is no equivalence relationship that gives directly

the abort semantics of this particular execution. But, by composing the two previously introduced conversations (Figure III.11 and Figure III.12) and their respective equivalence relationships, we can get the desired property. Indeed, two equivalent conversations may be substituted. The equivalence relationships are still valid because of the equality on the internal states. Thus, using the first equivalence relation stating the retry-ability, we can substitute one of the activity in the equivalence relation given for the atomicity property by the activity which executes the *Payment* operation several times. We can thus deduce an equivalence relationship between this activity and the activity *Login*, and hence that the activity would be rolled back.

III.4.3 Compensation and commutativity

Compensation is commonly used to undo the effect of an activity for long running activities. The main difference with an atomic conversation is that other conversations can be executed between the conversation that is to be rolled back and the compensation conversation, which can possibly change the internal state of the Web service. In the case of atomic execution, the abort operation or activity was to be executed right after the activity that should be cancelled.

Consider the activity depicted in Figure III.13 for the flight reservation Web service. The *Payment* operation is substituted with an equivalent *ComplexPayment* activity, which enables the service requester to pay part of the flight ticket price with its previously earned miles. For efficiency, the conversation is composed of two concurrent activities, one for paying by credit card and the other for paying with miles. The join condition on the final state ensures that the whole transaction terminates successfully only when both payment operations commit. Otherwise, if one of the payment fails, the other payment should be cancelled by invoking the corresponding cancellation operation. Furthermore, both payments may also be cancelled after a valid payment.

The following equivalence relations specify that a *CancelCreditCard* operation will cancel the credit card payment done with the operation *PayCreditCard*, and that the *CancelMiles* operation will cancel the payment with miles using the *PayMiles* operation:

```
<equivalence>
   <activity ref=''this:LoginCredit''/>
   <activity>
      <transition>
         <source activity=''this:LoginCreditCard''/>
```

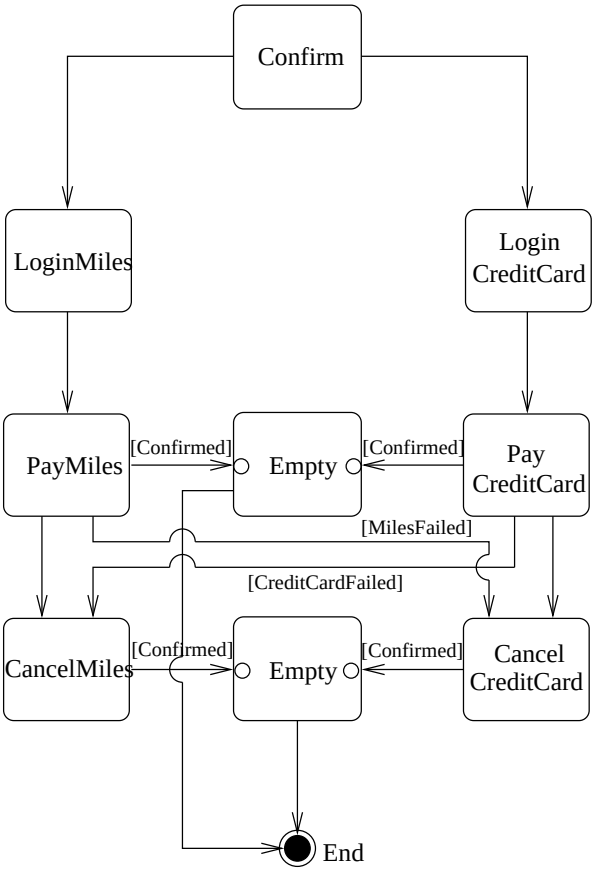


Figure III.13: Complex payment

```
            <destination state=''this:PayCreditCard''/>
        </transition>
        <transition>
            <source activity=''this:PayCreditCard''/>
            <destination state=''this:CancelCreditCard''/>
        </transition>
    </activity>
</equivalence>

<equivalence>
    <activity ref=''this:LoginMiles''/>
    <activity>
        <transition>
            <source activity=''this:LoginMiles''/>
            <destination state=''this:PayMiles''/>
```

```
        </transition>
        <transition>
            <source activity=''this:PayMiles''/>
            <destination state=''this:CancelMiles''/>
        </transition>
    </activity>
</equivalence>
```

However, this specification is not enough to fully describe compensation capabilities of the Web service. The service requester can, as it is permitted by the *concurrent* construct, call *PayCreditCard* and *PayMiles* operations sequentially. Thus, we can have the execution given by the sequence *Begin, PayCreditCard, PayMiles* and *CancelCreditCard*. For stating the compensation of the payment using the equivalence relationships given above, we need an additional property that states that the two payment operations are *commutative*. Based on this definition, we can substitute the two conversations to have the equivalent conversation, given by the execution path *Begin, PayMiles, PayCreditCard, CancelCreditCard*. Then, the equivalence relation stating the cancellation of the credit card payment can be used to derive that the payment is cancelled, despite the fact that the operation *PayMiles* is executed between the operations *PayCreditCard* and *CancelCreditCard*.

Commutativity of the above two activities of the flight Web service is given by the following equivalence relationship. Note that the two activities are both supported by the Web service because they can be simulated by the initial conversation.

```
<equivalence>
  <activity>
     <transition>
        <source state=''PayCreditCard''/>
        <destination state=''PayMiles''/>
     </transition>
  </activity>

  <activity>
     <transition>
        <source state=''PayMiles''/>
        <destination state=''PayCreditCard''/>
  </activity>
</equivalence>
```

III.4.4 Alternative conversations

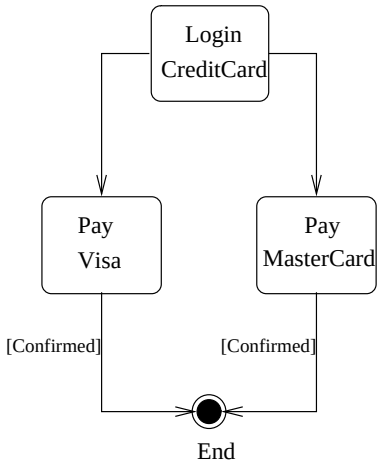


Figure III.14: Alternatives

Consider now the conversation depicted in Figure III.14, which specifies that there are two execution paths for making the payment, corresponding each to a different payment method. The difference between the previously introduced conversation is that we now have a choice construct instead of a concurrency construct. We can state that the two possible executions will lead to the same result by giving an equivalence relation between activities related to each execution:

```
<equivalence>
  <activity>
    <transition>
      <source state=''ws:LoginCreditCard''/>
      <destination state=''PayVisa''/>
    </transition>
  </activity>
  <activity>
    <transition>
      <source state=''ws:LoginCreditCard''/>
      <destination state=''PayMasterCard''/>
    </transition>
  </activity>
</equivalence>
```

This equivalence relation may then be exploited for implementing a specific recovery mechanism in case of failure of one operation: if one activity fails, the transaction can be retried with the alternative activity. A failure of one

activity may lead to apply compensation activities for restoring the state back before executing the alternative activity. Such a behaviour can easily be specified by expressing the recovery behaviour of each activity separately with several equivalence relationships similarly to those expressed in this section. The service requester can then compose different activities of the Web service to get the properties satisfying the requirements of the recovery mechanism it wants to apply.

III.5 Concluding remarks

As discussed in Section II.2.2, many conversation languages exist for specifying the externally observable behaviour of Web services. However, none of them provides all the necessary constructs for declaring the recovery support of Web services and their exceptional behaviour.

A service requester may need to invoke in parallel operations or conversations of a Web service for increasing performance. However, it should know exactly the concurrency behaviour of the Web service, in particular if it allows its operations or conversations to be invoked in parallel and under which conditions. Furthermore, the service requester can efficiently control concurrent accesses to the Web service if it knows how the operations of the Web service interfere. For controlling concurrent accesses from a service requester to a Web service, the concurrency behaviour should be defined at the service interface at the conversation level. This is specified in WS-RESC using the *concurrency* construct, coupled with the possibility to put join conditions on concurrent executions to synchronize concurrent activities. In addition, the correlation information specified on operations and on activities with WS-RESC enables identifying the support for concurrent sessions of a Web service.

There are many applications where meeting timing constraints is important for dependability. Given the timing constraints of a Web service and the timing requirements of the service requester, the service requester can select the Web service instance that best suits to it needs. The service requester can further invoke the Web service operations accordingly. In WS-RESC, it is possible to set timeouts individually on operations and to whole activities, and to specify what action should be performed if the activity timeouts.

The conversation language should allow defining exceptional behaviour by associating alternative conversations that should be executed when an exceptional event occurs, and should provide means to define exceptional events. In WS-

RESC, an exception is considered to be raised when a set of conditions are met, specified by the Web service developer and can include specific messages sent or received by the Web service, timeouts as well as non-WSDL messages such as HTTP or SOAP faults. Alternative activities are then specified for each of the exceptions, which are to be executed invoked by the service requester for handling the exception. WS-RESC further includes mechanisms for propagating exceptions across activities if an exception is not handled within an activity.

Finally, recovery properties should be expressed over conversations. The recovery properties must not be tight to a specific error recovery protocol. In addition, properties must be expressed in a formal way to allow reasoning about composition of conversations. Indeed, the WS-RESC description of a Web service can include several activities supported by the same Web service. However, a service requester may require interacting with the Web service following an activity that is not defined individually in the WS-RESC document, but that can be obtained as a composition of existing ones. The service requester can then identify the properties of composed conversations and verify if these properties match the requirements of a given error recovery protocol. Similarly to conversations languages that enable annotating conversations with pre-defined properties, conversations in WS-RESC can reference any property defined in an external document. However, the main feature of WS-RESC is to allow defining equivalence relationship between activities. The equivalence relation can then be used by the service requester to derive the properties of the activities. The formal definition of conversations and the equivalence relation further enables to service requester to compose the conversations and their properties.

Among the conversation languages examined in the context of this thesis none has the expressiveness of WS-RESC that can specify both the standard and exceptional conversations and includes constructs for specifying concurrency and timing constraints. The WS-RESC language enables expressing the recovery properties of the Web service, in a simple manner and not dependent to any specific recovery mechanism. Moreover, the compositionality feature of the conversations described in WS-RESC allows to derive properties of specific conversations not mentioned in the advertised interface but that are obtained by the composition of existing ones.

While the exceptional behaviour specification allows defining how to handle errors and the equivalence relation enables defining several properties that can be used for performing recovery actions, these are not sufficient to describe all the properties relevant to recovery because of the broad class of application-specific fault tolerance mechanisms. One restriction of our language is the abstraction from data, which reduces the expressiveness of the conversations. Indeed we can

specify the WSDL message names that should be received or emitted by the Web service but we cannot specify the values of the messages. Another issue shared with all conversation languages is to verify whether the Web service conforms to its conversation and to the related properties it declares or not. Verification at runtime can help to detect some incompatibilities but is not sufficient because the internal state of the Web service remains hidden. Complementary solutions related to trust management can then be integrated.

IV Dependable Composition of Web Services

This chapter proposes a new Web service composition language for building dependable composite Web services. The language supports concurrency with built-in mechanisms for controlling concurrent accesses to composed Web services at different isolation levels and enables specifying dependability mechanisms to be implemented both for forward and backward error recovery based on a coordinated exception handling model.

IV.1 Web Service Composition Actions (WSCA)

We define a composite Web service as a Web Service Composition Action (WSCA), which is specified by giving:

(1) The interface of the WSCA as a WSDL document.

(2) The specification of the WSCA including the behaviour of all the composite Web service's operations, using the WSCAL (Web Service Composition Action Language) declarative language.

The WSDL document is intended for service requesters of the composite Web service, while the WSCAL document is used for implementing the service. The internal structure of a WSCA is thus totally hidden to the outside world, and the WSCA composite Web service behaves as a standard Web service from that perspective.

Usage of WSDL and WSCAL documents for developing and deploying a composite Web service is illustrated in Figure IV.1. The WSCA, which is built according the WSCAL specification, is deployed on a centralized server and interacts with both service requesters and composed Web services. The application logic of

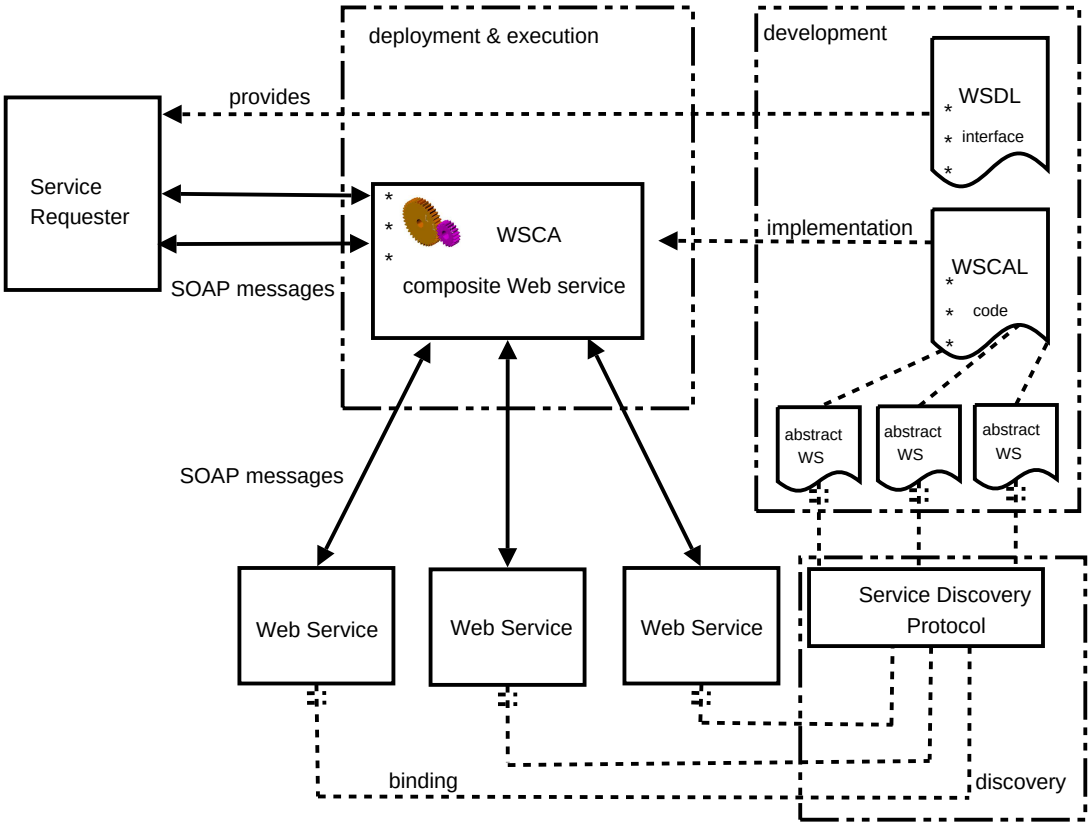
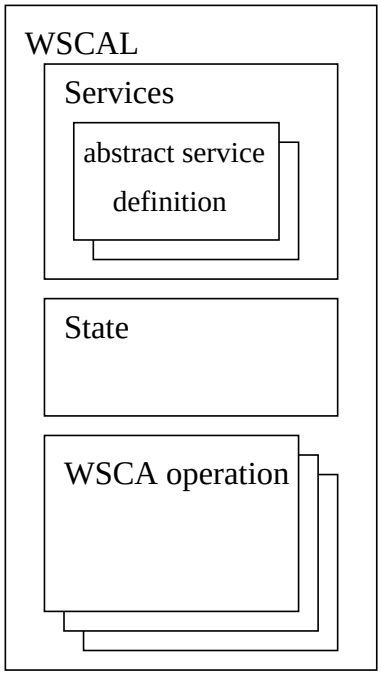


Figure IV.1: Web service composition actions

each operation offered by the composite Web service, which are declared in the WSDL document, should be specified in WSCAL. In the remainder of this document the term *WSCA operation* is used for referring to one operation of the composite Web service. Calls to composed Web services are specified as part of the application logic of the Web service operations in WSCAL. However, WSCAL allows referencing composed Web services abstractly, without giving the binding details of concrete services, which are discovered, thanks to a service discovery protocol, during the execution of a WSCA operation or prior to the execution, at deploy time. An executable composite Web service (the WSCA) may then be either implemented by the developer or generated from the specification, depending on the specific environment in which the service is to be deployed and the complexity of the service. Service requesters interact with the composite Web service directly, based on its interface definition given by the WSDL document.

IV.1.1 Specifying WSCA

The specification of a WSCA composite Web service is given using the WSCAL (Web Service Composition Action Language) XML-based language. A WSCAL document comprises three parts: i) an abstract declaration of composed Web services, ii) declaration of state variables shared among WSCA operations of the same WSCA instance, and iii), the behaviour of each WSCA operation (see Figure IV.2).




```
<WSCAL name=NCName ...>
  <services>
      (Detailed below)
  </services>
  <state>
      (Detailed below)
  </state>
  <WSCA operation=QName ...>*
      (Detailed below)
  </WSCA>
</WSCAL>
```

Figure IV.2: WSCAL document structure

A WSCA is defined using the $< WSCAL >$ XML element where the *name* attribute gives the name of the composite Web service. This top-level element comprises three parts: $< services >$ for declaring abstract definitions of composed Web services accessed during the execution of a WSCA operation, $< state >$ for declaring state variables of the WSCA instance and as many $< WSCA >$ elements as there are WSCA operations to be defined. The *operation* attribute references the Web service operation defined in the related WSDL document.

IV.1.2 Shared variables of a WSCA instance

Shared variables are accessible only by running WSCA operations belonging to the same WSCA instance. They are declared through the < *state* > element with the following structure:

```
<state>
   <types>
      <xsd:schema ... />*
   </types>
   <var name=NCName type=QName />*
</state>
```

The XML Schema data type system is used to specify new types within the < *types* > construct. Then, variables are declared using the < *var* > construct with an attribute *name* that defines a unique variable name visible within the WSCA and a *type* attribute for declaring its type.

IV.1.3 Abstract service definition

Composite Web services depend on autonomously deployed and administrated Web services, and therefore are very sensitive to the dynamics and evolution of the environment. One concern is the availability and reliability of composed Web services, which are crucial for guaranteeing the dependability of the provided composite service. If a single composed Web service becomes unavailable or starts to behave not as expected, the whole composite Web service can be affected. Furthermore, new Web services with new features (offering better performances, more reliable, trusty, secure etc.) can appear after the development phase of the composite Web service, and it would be beneficial to integrate them in the composition for a greater service quality. The specification of the composite Web service should thus take into account that composed Web services may change, more or less frequently.

As said earlier, we allow defining the composed Web services that are accessed in the composition process of a WSCA operation as abstract services. The developer specifies only the required interfaces of the composed Web service in terms of the provided operations, their messages and optionally, their conversation support. Then, actual Web services that are accessed are integrated using a service discovery protocol at deploy-time or at run-time, according to the WSCAL specification.

Concretely, the abstract definition of composed Web services is given through the < *services* > element. Then, the concrete binding with actual Web services may either be defined statically, by giving the endpoint addresses of services or, set dynamically at run-time, relying on a dynamic binding mechanism. The declaration of the < *services* > element is as follows:

```
<services name=NCName>
  <service name=NCName
      hrefSchema=anyURI
      conversation=anyURI ?
      strict=boolean:false
      isolation=QName?
  >*
    <staticService hrefSchema=anyURI /> *
    <dynamicService
        onCall=boolean:false
        multiple=boolean:false /> ?
  </service>
</services>
```

The abstract description of composed Web services is given in the *hrefSchema* attribute referring to the associated document. There is no strict requirement on how this document should be written, however it should be process-able by an underlying service discovery protocol. A typical description would be the abstract part of a WSDL definition including type information, operation names and message structures.

The optional *conversation* is used to reference a conversation description for declaring the interaction protocol required to be supported by the service, such as a document specified using the WS-Resc conversation language introduced in Chapter III. This document is then used by the service discovery protocol for finding Web services matching also the conversation support. The required conversation is also declared with the purpose of verifying correctness of interactions statically during implementation or dynamically at run-time. The attribute *strict* should be set to *true* if strict conformity to this conversation is required when interacting with the Web service, which can be in most cases detected and notified to the WSCA at run-time, or to *false* otherwise. The default value for this attribute is set *false* for performance issues. In that case, the conversation description is used only for discovering concrete services.

In our prototype implementation we use a service discovery protocol based on a partial matching of WSDL documents and on a simulation test for matching conversations expressed using WS-Resc (see Chapter V). In our protocol, the WSDL document of the discovered Web services should at least contain all the

elements defined in the document referenced with *hrefSchema*. In addition, if the conversation attribute is set, a simulation test is performed between the conversations supported by the Web services and the referenced document. A Web service then matches the abstract definition if it can simulate the required conversation support of the WSCA.

Each service may be statically bound to a specific Web service (defined by the $<$ *staticService* $>$ element) instance and/or dynamically bound to a Web service matching the abstract definition of the service interface that is given by the corresponding *definition* attribute (defined by the $<$ *dynamicService* $>$ element). In the former case, concrete binding information is provided through the WSDL document associated with the Web service. In the latter case, a matching Web service is located at runtime using a service discovery protocol as discussed above. Dynamic binding of composed Web services matching abstract Web service definitions may take place either upon first invocation of the composed service or upon instantiation of the composite Web service, according to the value of the *onCall* Boolean attribute of the given service. The default value for this attribute is set to *false*. Composed Web services bound to a specific WSCA instance are kept during the life-time of the WSCA instance. In a different instance, the service discovery protocol can select, in general, different instances for the composed Web services.

For the sake of availability, we allow each composed service to be bound to a set of Web services all matching the specification of the associated abstract service instances; this is specified using the *multiple* Boolean attribute in the $<$ *dynamicService* $>$ element for which the default value is *false*, and by stating as many Web services as required with the $<$ *staticService* $>$ element. Then, a unique Web service instance that is available among all bound instances is chosen for the whole WSCA instance at the first invocation of the corresponding service. If the Web service instance becomes unavailable later during the execution, an alternate instance can be used if the call to the Web service operation is specified as retry-able on an alternate Web service by the developer, in the WSCAL specification. Otherwise, an exception is raised and another Web service instance can only be chosen if it is explicitly specified in the corresponding handler (see definition of the *call* statement at page 105).

As an illustration, a sample of the $<$ *services* $>$ element for the travel agency composite service is given below. The service offers the *JointBooking* WSCA operation that coordinates booking over hotel and flight booking Web services, for which concrete Web services are dynamically retrieved upon invocation of the WSCA operation. The composed Web services are required to have abstract service interfaces described respectively in the *Flight.wsdl* and *Hotel.wsdl* WSDL

documents where only messages and operations are defined without concrete binding information, and should be able to implement the conversations defined respectively in the *Flight.resc* and *Hotel.resc* documents.

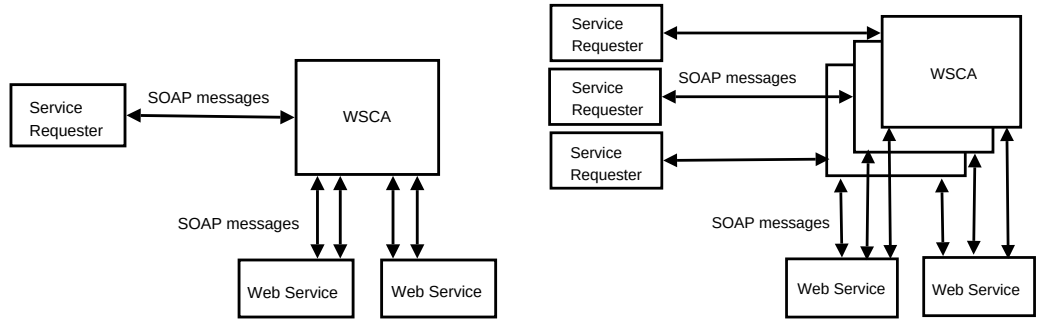
We get:

```
<services>
  <service name=''FlightService''
        hrefSchema=''ta.com/Flight.wsdl''
        conversation=''ta.com/Flight.resc''>
    <dynamicService
        multiple=true
        strict=true />
  </service>
  <service name=''HotelService''
        hrefSchema=''ta.com/Hotel.wsdl''
        conversation=''ta.com/Hotel.resc''>
    <dynamicService
        multiple=true
        strict=true />
  </service>
</services>
```

Interactions with composed Web services are defined in the specification of WSCA operations. In particular, the same Web service instance can be accessed within different WSCA operations of the same composite Web service. This may lead to concurrent accesses to composed Web services when service requesters call concurrently the WSCA operations. These concurrent accesses then might need to be controlled for the consistency of the WSCA composite Web service.

IV.1.4 Concurrency

Composed Web services are autonomously administrated open systems that can be accessed concurrently by several autonomous service requesters. A service requester is in general not aware of the existence of other concurrent calls to the Web service from other service requesters and the Web service is the only responsible for controlling concurrent accesses to it for maintaining consistency. However, concurrent accesses to composed Web services originating from the same WSCA can be problematic as it is dependent on the concurrency control of the composed Web services, which can be unknown or non-compatible with each other. Concurrent accesses should thus be controlled at the WSCA-side with built-in mechanisms without tight coupling with the concurrency control of composed Web services.



(a) Intra WSCA concurrency. Composed Web services are accessed concurrently from a single WSCA instance.

(b) Inter WSCA concurrency. Composed Web services are accessed concurrently from several WSCA instances.

Figure IV.3: Concurrent accesses to composed Web services

Two types of concurrency issues arise in the execution of a WSCA operation. The first issue is internal to a single WSCA operation. It concerns the control of concurrent accesses to composed Web services during the execution of the WSCA operation (referred to as intra-WSCA concurrency, see Figure IV.3(a)). The second issue concerns concurrent accesses to composed Web services as a result of concurrent invocations of WSCA operations accessing the same Web services (referred to as inter-WSCA concurrency, see Figure IV.3(b)). While for controlling inter-WSCA concurrency we rely on traditional solutions based on session management, the intra-WSCA concurrency is controlled using nested actions that executes in isolation from each other. We first define the concurrency support for concurrent WSCA executions for controlling inter-WSCA concurrency. Intra-WSCA concurrency is further addressed in the next section.

In the WSCAL specification, the behaviour of the concurrent WSCA executions are set by two parameters. First, the developer specifies how a WSCA is instantiated when a service requester access one WSCA operation and the life-time of the service instance. Similarly to existing solution in the area, three types of service instantiation is possible:

(1) *Request*: a new service instance is created for each service request and terminates when the interactions of the WSCA operation terminate.

(2) *Application*: a service instance is created at the first invocation of the service, and all subsequent invocations from the same or different service requesters interact with this instance.

(3) *Session*: a different service instance is created for each service requester, and all subsequent operation requests of service requesters go to the associated service instance.

When two service requesters interact with a single instance, the state variables of the WSCA are shared between the two executions and all bindings with composed Web services are preserved.

The service instantiation parameter applies to the whole WSCA, no matter the different concurrency features of composed Web services. It is specified using the *scope* attribute of the top-level *WSCAL* element defining a WSCA:

```
<WSCAL name=NCName
       scope=request | application | session:request>
  ...
</WSCAL>
```

The second parameter specifies how concurrent accesses to composed Web services resulting of concurrent calls of WSCA operations by service requesters are controlled.

In database systems the behaviour of transactions when accessing shared data is controlled by a particular locking strategy according to isolation levels. The higher is the isolation level, the more longer locks are maintained. The American National Standards Institute (ANSI) defines four isolation levels [ANSI, 1992]. The highest isolation level *serializable* (level 3) locks all shared data for both reading and writing until the end of the transaction, and the lowest isolation level *read uncommitted* (level 0) locks only shared data that are modified and keeps the lock until the end of the transaction. Lower level isolation levels are used in particular for improving performance by increasing concurrent accesses at the cost of consistency. In the context of Web services we can not use the ANSI isolation levels as is because they assume that operations are read or write operations, while semantics of operations on composed Web services can not been determined in general. Similarly to the ANSI isolation levels, we set isolation levels for controlling access to Web services, which are seen as shared resources. However, contrary to database systems, we do not lock the resources but restrict accesses at the client-side. Three types of isolation levels are identified:

(1) *none*: this is the default isolation level, which means that accesses are not controlled, and any execution of a WSCA operation can access the composed Web service.

(2) *WSCA-visible*: composed Web services that are WSCA-visible can be accessed only by a unique WSCA instance. Accesses from other WSCA instances (for the same or different WSCA operation) have to be delayed until the termination of the ongoing interaction.

(3) *strict*: Concurrent accesses to composed Web services are only allowed within a single execution of a WSCA operation.

Contrary to the first parameter, which is specified once and applies to all WSCA operations, a different isolation level can be specified for each of the composed services. The isolation level is specified using the *isolation* attribute of the $< service >$ element defining an abstract composed Web service. Note that different concurrency control techniques, according to the composed Web services concurrency supports, can be implemented for guaranteeing the above rules and increasing the overall performance. Different solutions based on the exploitation of the concurrency support of individual Web service expressed in WS-RESC are presented in Chapter V. In particular, for increasing performance by not delaying concurrent calls from different composed Web services when a *strict* isolation level is set, we allow concurrent calls if the composed Web services accessed within the WSCA operations support session-based interactions.

The above isolation levels control the concurrent accesses to composed Web services from different composite Web service instances. At any isolation level, concurrent accesses are not controlled during the execution of a single WSCA operation, which is the focus of the next section.

IV.2 Coordinating access to composed Web services

This section deals with the coordination of accesses to composed Web services within an execution of a single WSCA operation. The main objective is to establish a coordination model suited for expressing the composition of Web services using different fault tolerance mechanisms. The approach is based on the coordinated atomic actions model, which provides a base structuring model for developing fault tolerant systems by combining both backward and forward error recovery mechanisms. The base CA actions model is adapted to the specifics of Web services, in particular by relaxing isolation and atomicity requirements over interactions with composed Web services and introducing a new coordinated exception handling model. We first present Coordinated Atomic Actions, which inspired our work. Then, we detail our approach in the subsequent sections.

IV.2.1 Coordinated Atomic Actions

Coordinated Atomic Actions (CA Actions) [Xu et al., 1995] are a structuring mechanism for developing dependable concurrent systems through the generalization of the concepts of atomic actions [Campbell and Randell, 1986] and transactions [Gray and Reuter, 1993]. Atomic actions are used for controlling cooperative concurrency among a set of participating processes and for realizing coordinated forward error recovery using exception handling. Transactions are used for maintaining the consistency of shared external resources that are competitively accessed by concurrent actions. Each CA Action is designed as a multi-entry unit with roles activated by action participants, which cooperate within the action. A transaction is started on external objects and it terminates at the end of the CA Action.

A CA Action terminates with a normal outcome if no exception has been raised or if an exception has been raised and handled successfully; all transactions on external objects are then committed. If a participant raises an exception inside an action and if the exception cannot be handled locally by the participant, the exception is propagated to all the other participants of the CA Action for *coordinated error recovery*. If several exceptions have been raised concurrently they are resolved using a resolution tree imposing a partial order on all action exceptions, and the participants handle the resolved exception [Campbell and Randell, 1986]. If coordinated recovery fails, the Coordinated Atomic Action terminates with an exceptional outcome. An exception is then signaled by the CA Action and transactions on external objects are aborted.

Coordinated Atomic Actions can be designed in a recursive way using action nesting. Several participants of a CA Action can co-enter into a *nested CA Action*, which defines an atomic operation inside the embedding CA Action. Accesses to external objects within a nested action are performed as nested transactions so that if the embedding CA Action terminates exceptionally, all sub-transactions that were committed by nested actions are aborted as well. A CA Action participant can only enter one concurrent nested action at a time. Furthermore, a CA Action terminates only when all its nested actions have completed. Note that if the nested action terminates exceptionally, an exception is signaled to the containing CA Action.

As an illustration, Figure IV.4 depicts the execution of a CA Action that is composed of three participants *P1, P2*, and *P3*. A nested action is created with participants *P1* and *P2*; nested transactions are further executed on external objects. An exception raised by participant *P3* is propagated to participants *P1*

and *P2*, which causes the CA Action to enter an exceptional state, as shown by the greyed box, where the participants cooperate for handling the exception.

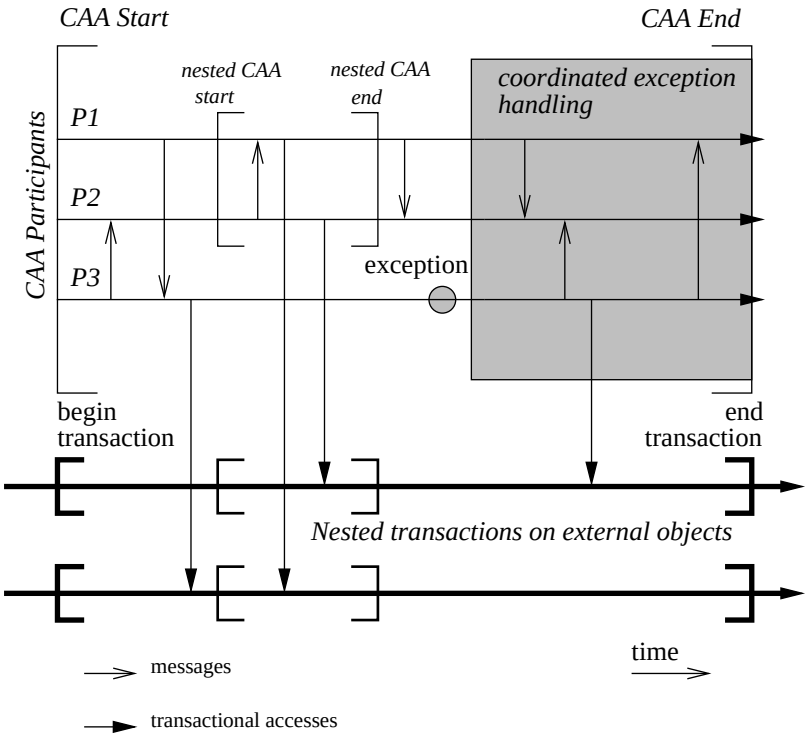


Figure IV.4: Coordinated atomic actions

CA Actions mainly focus on structuring concurrent systems and on providing their fault tolerance by exception handling. One of the main intentions behind CA Actions is to employ them as a core mechanism for structuring complex distributed applications: they promote recursive view on system execution by abstracting away both normal and abnormal behaviour of the low level software. A formal specification of CA actions can be found in [Tartanoglu et al., 2003b, 2004].

To deal with backward error recovery, CA actions are based on a nested transactional model that coordinates transactional accesses to external resources. However, as raised previously, ACID properties over external resources are not suited in the case of Web services. Strict requirements over external resources can be relaxed for Web services. Indeed, atomicity is not always required or can be obtained using advanced transactions based on open-nested transactions with compensation operations, which relax isolation. Concurrent accesses to Web services can be coordinated by relying on the support for concurrency of individual Web services or by controlling concurrent accesses at the client side.

The forward error recovery mechanism of CA actions is based on the coordinated handling of exceptions by all the participants of the CA action. Occurrence of concurrent exceptions then implies resolving the concurrent exceptions into a single exception for coordinated error recovery. However, due to the autonomy and to the lack of transactional support of Web services, actions done on composed Web services can not always be undone and specific actions may be required to perform on accessed Web services when failures occur. Concurrent exceptions may thus imply executing several exception handlers for safely terminating interactions with different Web services. We therefore extend the coordinated exception handling model by allowing several exception handlers to be executed when concurrent exceptions occur to deal with the diverse exceptional behaviours of composed Web services.

IV.2.2 WSCA operations

Based on the CA Action model, a WSCA operation is defined as a process that comprises several threads of execution called *participants*, which execute concurrently. During execution, participants can co-operate by sharing information using local variables and synchronizing their execution. Each participant interact further with one or more composed Web services. Contrary to CA actions we do not impose transactional accesses to composed Web services. Action nesting is used for controlling concurrent accesses to composed Web services and for defining coordinated exception handling scopes. A broad range of autonomous Web services can thus be integrated in the composition, despite their lack of support of transactional properties. Furthermore, the design of a composite Web service with several participants allows to easily implement a choreography (see Chapter II) by assigning a different participant for each of the roles defined in the choreography.

Input messages received by the WSCA for a particular WSCA operation (action for short) lead to the synchronized execution of the action participants. The input message can be read by all the participants, which can further access to a set of variables shared among all the action's participants (complementary to shared variables of the WSCA instance). The action terminates synchronously when all the participants terminate their computation. According to the message exchange pattern defined in the WSDL document of the WSCA, messages defined as output or fault messages are computed (explained later) and sent to the service requester at the end of the action. If the action terminates normally, WSDL output messages will be returned. If the action terminates exceptionally, i.e., if an unhandled exception occurs or if the handling of an exception causes

the action to abort, a fault message is returned.

The call of a WSCA operation from a service requester does not lead systematically to the creation of a new WSCA instance. As said in the previous section, a new instance is created for each operation call when the scope attribute of the *WSCA* is set to *request* or for each different sessions (in general for each service requester) when the scope is set to *session*. Shared variables of a WSCA are initialized at the first invocation of a WSCA operation and their values are maintained for subsequent calls of other WSCA operations. Similarly, binding with Web services are maintained during the life-time of a WSCA instance. State variables for local computations within a WSCA operation can also be declared. These local variables are however re-initialized at each invocation of the WSCA operation.

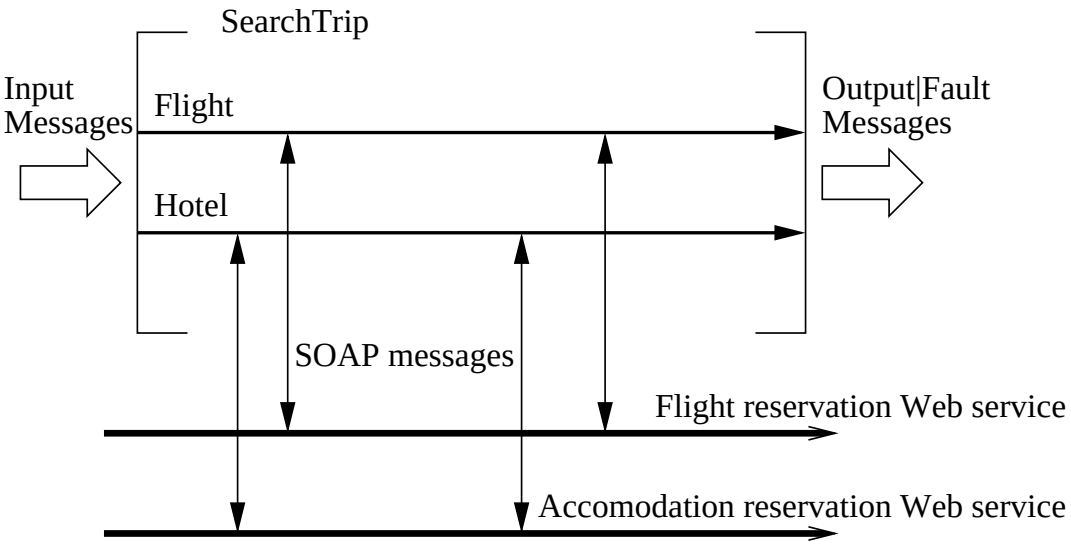


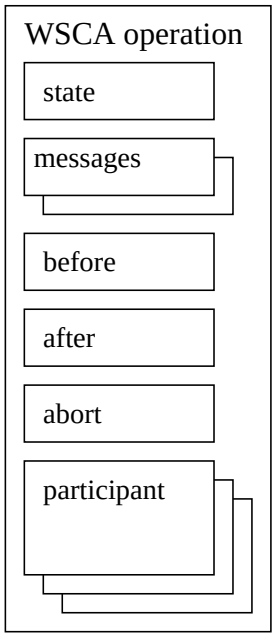
Figure IV.5: Execution of a WSCA operation

Figure IV.5 illustrates the execution of the *SearchTrip* WSCA operation for the travel agency composite Web service. The *SearchTrip* operation is called with an input message comprising the details of the requested trip. The WSCA then contacts composed Web services and returns either an output message comprising the list of trips or a fault message if no trip for this request can be found. The WSCA operation comprises two participants. The participant named *Flight* interacts with a flight reservation Web service and the other one, named *Hotel*, interacts with an accommodation reservation Web service. The two participants execute independently and synchronize for terminating. A unique response is then computed and returned to the service requester. If both reservations have succeeded, their results are combined for generating a list of trips, which is re-

turned as an output message to the service requester. Otherwise, a fault message is returned informing the service requester that no trip is found.

The behaviour of a WSCA operation is defined in WSCAL using the $< WSCA >$ element (see Figure IV.6). The *operation* attribute gives the name of the composite Web service operation that is specified. It references the operation declared in the associated WSDL document of the WSCA. The *exceptionRules* attribute is used for referencing a document used to select the exception handlers to execute when several exceptions are raised concurrently (see Section IV.2.4).

The declaration of a WSCA operation comprises three parts. The first part, which comprises the *input, output, fault* and *state* elements, is for general declarations that concern the whole action participants such as the declaration of local variables. The second part comprises the *before, after* and *abort* elements and specifies respectively actions performed before and after the execution of participants and the actions performed in case of exceptional termination.




```
<WSCA operation=QName
      exceptionRules=anyURI
   <input message=QName  name=NCName>*
   <output message=QName  name=NCName>*
   <fault message=QName  name=NCName>*
   <state>?  . . .  </state>
   <before>  . . .  </before>
   <after>  . . .  </after>
   <abort>  . . .  </abort>
   . . .
   <participant name=NCName>*
      . . .
   </participant>
</WSCA>
```

Figure IV.6: WSCA operation construct

Input, output and fault messages used for interacting with the service requester are assigned to variables defined respectively in the $< input >$, $< output >$ and $< fault >$ elements. The *message* attribute references the associated message defined in the WSDL document and the *name* attribute defines the local variable name to which the content of the message will be assigned and which is to be used in local computations.

Additional local variables, seen only from the action participants of a single WSCA operation call are declared through the < *state* > defined as follows, similarly to the declaration of shared variables of a WSCA composite Web service (see Section IV.1.2).

```
<state>
   <types>
      <xsd:schema ... />*
   </types>
   <var name=NCName type=QName />*
</state>
```

Local variables of the WSCA operations can be accessed by all the action participants. However, accesses from participants that entered a nested action follow nested transaction rules (see Section IV.2.3 for details).

The $< before >$, $< after >$, $< abort >$ blocks are used to specify, which actions are performed when initiating, terminating and aborting a WSCA operation. The *before* construct is executed at the first invocation of the WSCA operation, before starting the execution of participants. It can be used for example to assign initial values to local variables according to the input parameters of the WSCA operation. The *after* pattern is executed when all participants have terminated their respective executions. It is commonly used to specify the computation of WSDL output messages to be returned to the service requester over the results of each individual participant. The *abort* pattern is executed when the WSCA terminates exceptionally and specifies the exception to be signaled to the outside, by assigning appropriate values to the WSDL fault messages.

Calls to composed Web service operations are declared in the control flow of each participant. Any participant can access freely all composed Web services declared in the < *services* > section of the WSCAL document. However, depending on the needs of the composite Web service application, the developer can enforce isolated accesses to some composed Web services using action nesting.

IV.2.3 WSCA nesting

Control of concurrent accesses to composed Web services within a WSCA operation call is governed according to the isolation level specified at the level of the WSCA, with the *isolation* attribute of a < *service* > element. Different isolation levels, which apply for all the operations of the WSCA, can thus be set for different composed Web services. However, these rules do not specify the control for concurrent accesses within a single WSCA operation execution but

for concurrent accesses within different WSCA instances. The default concurrency behaviour, whatever is the isolation level, is that all accesses to composed Web services are permitted from participants of a WSCA operation. The general assumption behind this decision is that Web services are autonomous systems responsible for maintaining their own internal consistencies. Isolation of different interactions with a single or multiple service requester is guaranteed by the Web service. However, the service requester often maintains information used by the Web service to track different sessions, such as HTTP session cookies or specific values exchanged in each sent and received messages. Participants sharing this kind of information can thus lead to inconsistencies when accessing the same composed Web service, by interfering their own independent interactions. We allow thus to define a finer grain control over concurrent accesses to composed Web services using action nesting, by isolating accesses to composed Web services within nested actions.

WSCA operation nesting (*action nesting* for short) is similar to that of CA Actions [Xu et al., 1995] and is used as a structuring technique for controlling concurrent accesses to composed Web services within an execution of a WSCA operation. Indeed, during the execution of a WSCA operation, a subset of the action's participants can join together to form a nested WSCA, which is viewed as an isolated operation from the standpoint of the containing action: participants involved in a nested action can neither *communicate* with other participants outside the nested action, nor they can join sibling nested actions.

Nested actions are pre-defined with a static list of the involved participants. Each participant then specifies in its control flow when it wants to enter a specific nested action, which executes when all its participants have joined. Similarly, the nested action terminates when all the participant involved in the nested action terminate their respective executions.

Accesses to composed Web services from top-level and nested actions are isolated from each other. In our implementation, this is ensured through a concurrency manager located on the WSCA platform (see Chapter V). Note that controlling concurrency do not rely on any transactional protocol support of the composed Web service, which are autonomous entities. Therefore only accesses within a single WSCA instance are controlled. In general, when a nested action is created, the accesses to composed Web services from the participants of the parent action are delayed until the termination of the nested action. However, more flexible mechanisms allowing concurrent accesses could be implemented for increasing performance, which requires knowledge of the support for concurrency of composed Web services.

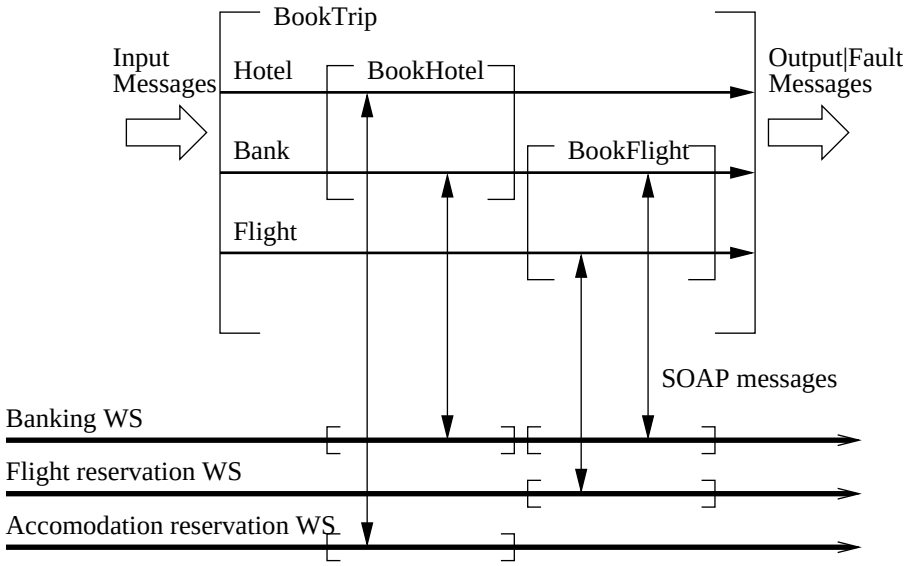


Figure IV.7: Nested WSCA execution

Figure IV.7 depicts a WSCA operation execution for the *BookTrip* operation of the travel agency composite Web service. The *BookTrip* action is defined with three participants, *Flight*, *Hotel* and *Bank*. Operations done on the banking Web service should be isolated since they access to a unique bank account. Therefore two nested actions are defined, one for the booking of the hotel room (*BookHotel*) and another for the booking of the flight seat (BookFlight). The figure represent one possible execution, where the two nested actions are executed sequentially with the execution order *BookFlight* and then *BookHotel*. However, if the nested actions are called concurrently, the order *BookHotel*, *BookFlight* could also be possible. Furthermore, if the Banking Web service can distinguish different sessions from a single service requester, accesses can be done concurrently to improve performance by not delaying the execution of one nested action. Such an optimization based on the online analysis of the concurrency support expressed in WS-RESC of discovered Web services is presented in Chapter V.

Local variables of a parent action can be accessed concurrently by the participants of the parent action and by the participants of all nested actions. Shared variables of the WSCA can be accessed within different WSCA operation executions on the same WSCA instance. Therefore, accesses to those shared resources should be isolated to guarantee consistency. Participants of a nested action make thus transactional accesses to the shared variables of the WSCA and to the local variables of their parent action, following nested transaction rules. If the nested action terminates exceptionally, all operations done on the transactional vari-

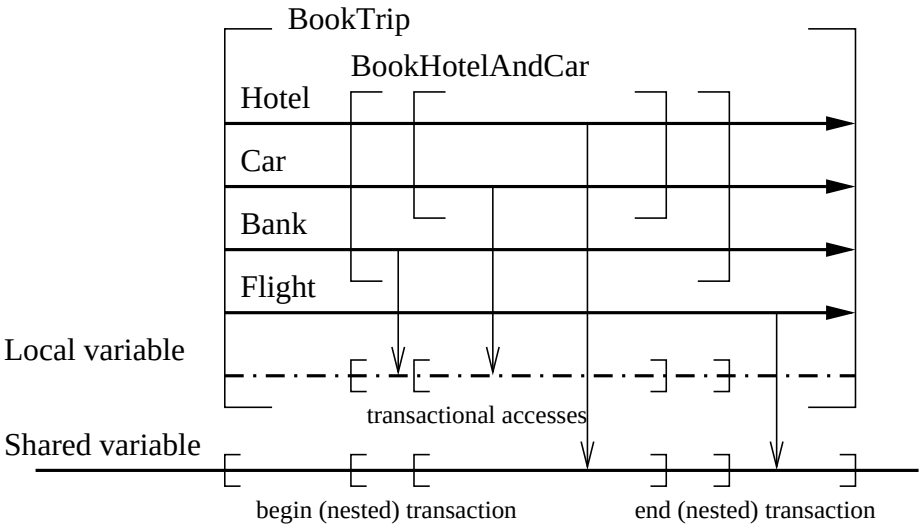


Figure IV.8: Transactional accesses to shared and local variables

ables are aborted. The updates of the values of the variables can be seen by participants outside the nested action only when the latter terminates normally and commits its operations. Figure IV.8 illustrates the nested transactions performed on local and shared variables, where brackets are used to delimit nested transactions.

In WSCAL, we introduce the $< nestedWSCA >$ element for describing a nested action. For each WSCA operation, all the nested actions are first declared with the list of composed Web services that will be accessed, the actions executed before, after and when aborting the nested action, the names of the participants that are involved in the nested action and the recursive definition of the nested actions in the nested action. The behaviour of participants of a nested action are further specified within the control flow statements of each participant. The structure of the $< nestedWSCA >$ element is as follows:

```
<nestedWSCA name=NCName
            exceptionRules=anyURI>
   <state>
      Local variables
   </state>
   <before> ?
      Statements
   </before>
   <after> ?
      Statements
   </after>
   <abort> ?
```

```
        Statements
    </abort>
    <participant name=QName />*
    <nestedWSCA name=NCName>
        Recursive definition
    </nestedWSCA>
</nestedWSCA>
```

Even though isolation of accesses to composed Web services are enforced using nested actions, the lack of transactional support of composed Web services makes difficult the proper reasoning about the behaviour of composed Web services when faulty interactions occur. We rely on a forward error recovery model for specifying application-specific recovery strategies. The model defines how exceptions are propagated across WSCA participants and nested WSCAs and the coordinated handling of exceptions among action participants.

IV.2.4 Coordinated exception handling

The error recovery model of WSCA is based on coordinated exception handling, which can be used to specify both backward and forward error recovery mechanisms. Coordinated handling of exceptions within WSCAs follows the exception handling model of atomic actions [Campbell and Randell, 1986]. However, we adopt a different resolution algorithm for the handling of concurrently raised exceptions, more suited in the context of Web services. Indeed, a composition process involves multiple interactions with third party Web services and interrupting an established connection or terminating prematurely a complex interaction sequence is not always desired. Furthermore, the exceptional behaviour will vary according to different Web services that fail and can require specific interactions with the service requester.

During the execution of a WSCA operation, exceptions can be raised explicitly using a language construct, or they can be raised automatically by the underlying runtime system. The exceptional behaviour of a WSCA operation follows a multi-level scheme: (i) local handling of the exception at the participant level, (ii) propagation of unhandled exceptions to all participants of WSCA operations or of nested WSCAs for coordinated handling, and (iii) exception signaling to service requesters or containing actions when coordinated exception handling fails.

Each participant defines its exception handling contexts and when an exception is thrown within an exception handling context the participant suspends its

execution. An appropriate exception handler is executed, which terminates the execution of the exception handling context. Exception handling contexts can be nested. In this case, if an exception is raised and a handler is not defined within the context, the exception is propagated to the containing context until a matching exception handler is found. A default exception handler can also be defined, which catch all exceptions. Similarly, exception handlers can define recursively exception handling contexts for exceptions occurring during the execution of the handler.

If no matching handler for an exception can be found, and if a default exception handler is not defined the exception is propagated to all the participants of the WSCA operation for coordinated recovery. If the participant, which raised the exception, is involved in a nested action, the exception will be propagated to all the participants of the nested action only. All participants that are informed of the propagated exception then suspend their execution and each of them execute synchronously a coordinated exception handler. Note that the participant, which is notified of a propagated exception, interrupts its execution safely, by raising an internal exception. In a prototype WSCA implementation (see Chapter V), participants check the presence of a propagated exception by accessing a controller object before entering a nested action, calling a composed Web service, synchronizing with another participant and terminating actions. If an exception is propagated, then the participant raise explicitly a local exception. The exception handling context for the propagated exception is the whole WSCA operation or the nested WSCA if the exception occurs in a nested action. Therefore, when all coordinated handlers terminate, the WSCA operation or the nested WSCA will terminate its execution.

If some participants are involved in a nested action and an exception is propagated from a participant not involved in the nested action, then a special *abort* exception is raised within the nested action so that the nested action terminates safely (e.g., by terminating interactions with composed Web services). After that, the propagated exception will be raised on all participants that have finished their nested action.

When all participants of the WSCA operation or of the nested WSCA have caught the propagated exception, they execute their respective exception handlers. Participants can define exception contexts within exception handlers to catch new exceptions occurring within the handler. However, if there is an unhandled exception that is raised, it is not propagated to other participants but a special *failure* exception is thrown, and the participant suspends its execution. Other participants continue their execution uninterrupted.

When all participants have terminated their exception handlers, the WSCA operation or nested action terminates with several outcomes. If no *failure* exception has been raised by any participant, the action terminates normally and the *after* block of the action is executed. The *after* block can in particular be used to compute an output message to be sent to the service requester in case of a top-level WSCA operation. If at least one participant raised a *failure* exception, the *abort* block is executed. If the action is a top-level WSCA operation, the *abort* block should compute the WSDL fault messages to be sent to service requesters. If it is a nested action, a unique exception should be raised to signal the exception to the containing action. The signaled exception is thrown locally on each of the participants that were involved in the nested action.

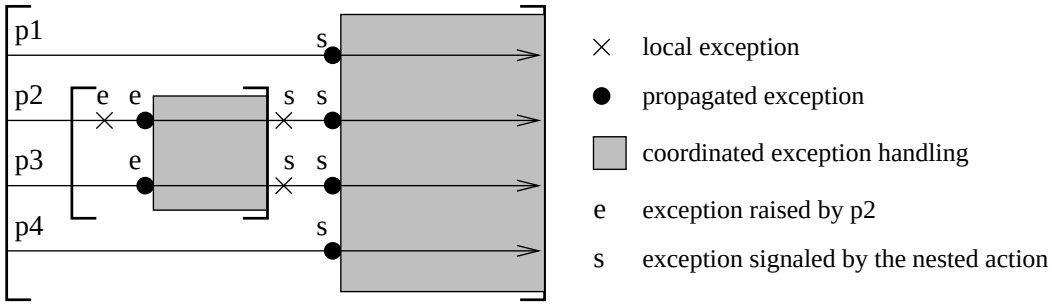


Figure IV.9: Coordinated exception handling in a WSCA operation

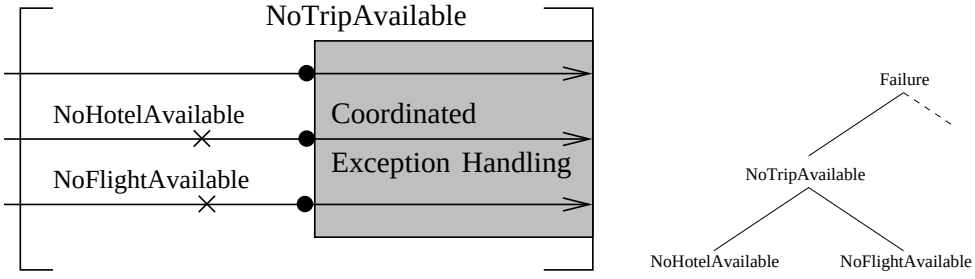
Figure IV.9 illustrates the exception handling model of WSCAs. Participant *p2* raises the exception *e* inside a nested action involving besides itself, the participant *p3*. The exception is not handled by *p2* and is immediately propagated to the participants of the nested action. *p2* and *p3* then synchronously execute their respective coordinated exception handler for the exception *e*. Coordinated handling fails, and the nested action terminates exceptionally, leading an exception *s* be signaled at the level of each participant involved in the nested action. Similarly, the signaled exception is first tried to be handled locally to participant, and on failure, it is propagated to all the participants of the WSCA operation for coordinated recovery.

IV.2.5 Concurrently raised exceptions

During the execution of a WSCA operation, different participants can raise exceptions concurrently. If more than one exception are propagated for coordinated recovery before the participants synchronize for starting their exception handlers, there is a concurrent exception issue. Indeed, exceptions will be propagated to

participants potentially in different orders, and there must be a consensus among participants about the exceptions to handle. Different approaches exist for dealing with concurrently raised exceptions. One of the exception can be chosen among all propagated exceptions and the other ones be discarded. Or, a unique exception, and possibly different from all that were raised, can be raised for handling all the errors. Another solution is to consider all the exceptions in a specific order and handle all of them separately.

The major difficulty in adopting one concurrent exception resolution strategy is that most of the exceptions would be a result of a failure in the interaction with a composed Web service. Given that interactions are not transactional, there might be specific actions to do for recovering the error on that particular Web service. Discarding exceptions as in the case of the first approach would thus be a cause of inconsistency as it would leave faulty Web services. The second approach, which is the approach that is adopted in the coordinated exception handling of atomic actions [Campbell and Randell, 1986], is based on the assumption that a unique handler, which can also be a union of different handlers, would be sufficient for handling all the exceptions. The exception resolution algorithm is based on a rooted exception tree containing all expected exceptions and a failure exception at the root [Xu et al., 2000]. Figure IV.10 illustrates the resolution of two concurrently raised exceptions *NoFlightAvailable* and *NoHotelAvailable* into a single exception *NoTripAvailable* that is jointly handled by all participants of the WSCA operation. The corresponding exception tree used to resolve the concurrent exceptions is given in Figure IV.10(b).



(a) Concurrent occurrences of NoHotelAvailable and NoFlightAvailable exceptions

(b) The exception tree

Figure IV.10: Concurrent exception resolution into a single exception

The approach to use an exception resolution algorithm based on an exception tree to compute a single exception enables an efficient exception handling model since only one exception is to be cared of. However, in the context of Web service composition, all the composed Web services have different exceptional behaviours, resulting in a large number of different exceptions. Therefore, building

an efficient exception tree for all the exceptions is not straightforward. Furthermore, some Web services may require specific conversations to be executed when failure occurs and writing an exception handler addressing the needs of failures of multiple Web services is not feasible.

This is why we consider the third approach, which allows to execute several exception handlers. The Guardian model [Miller and Tripathi, 2002] for exception handling is based on this approach. The resolution algorithm of this model computes a sequence of exception handlers to be executed in a given order when concurrent exceptions are raised. Different rules can be applied for selecting the execution sequence of handlers, such as random choice, first come first served or priorities.

In WSCA, we define a resolution function that, based on a set of exception resolution rules associated with each WSCA operation and each nested WSCA, decides which exceptions should be handled. Furthermore, only participants, which have defined a handler for these exceptions execute the exception handlers, while others wait suspended until the end of the coordinated exception handling process. In the absence of resolution rules, or if the rules does not contain a rule for a particular set of exceptions, then all the associated handlers of concurrent exceptions are to be executed. If there are exception handlers involving distinct set of participants, these handlers can be executed concurrently. Otherwise, for coordinated exception handlers with at least one common participant, handlers should be executed sequentially. The sequence can be determined with the exception resolution rules. If no sequence has been defined, a random order is chosen. However, the order is always the same for all the participants, which start and terminate the execution of handlers for the same exception synchronously.

The exception resolution document defines the rules for the resolution of concurrently raised exceptions. A < *rule* > element contains a < *concurrent* > element for declaring the set of exceptions and a < *resolution* > element that gives the list of exceptions to be handled. Exceptions that must be handled in a particular sequence are specified within a < *sequence* > construct. Exceptions are referenced using the *ref* attribute of the < *exception* > element. The reference *any* can be used within a < *concurrent* > block to reference all exceptions. We get:

```
<rule>
   <concurrent>
      <exception ref=QName/>*
   </concurrent>
   <resolution>
      <sequence>*
         <exception ref=QName/>*
```

```
        </sequence>
        <exception ref=QName/>*
    </resolution>
</rule>
```

If there is a < *concurrent* > block matching all concurrently raised exceptions then the associated < *resolution* > is chosen. Otherwise, all partially matching blocks are considered and their < *resolution* > blocks are concatenated. Furthermore, if no rule matches a particular exception, it is also added to the list of exceptions to be handled. For example, if there is a rule for the concurrent occurrences of exceptions *e1* and *e2* and if three exceptions *e1, e2* and *e3* are raised concurrently, then the rule for *e1* and *e2* would be applied and the exception *e3* will be added to the list of the exceptions to handle.

The example exception resolution document below defines two rules for the *BookTrip* WSCA operation of the travel agency composite Web service. The first rule specifies that if *HotelBookingFailed* and *FlightBookingFailed* exceptions are raised, then, both exceptions should be handled with the flight exception first. The second rule specifies that if there is a *PaymentError* exception among all concurrently raised exceptions, then the new *Abort* exception should be raised in all participants and handled. If all three exceptions are raised concurrently, according to the definition given above, only the second rule would be applied as it is a perfect match.

```
<rule>
    <concurrent>
        <exception ref=this:HotelBookingFailed/>
        <exception ref=this:FlightBookingFailed/>
    </concurrent>
    <resolution>
        <sequence>
            <exception ref=this:FlightBookingFailed/>
            <exception ref=this:HotelBookingFailed/>
        </sequence>
    </resolution>
</rule>

<rule>
    <concurrent>
        <exception ref=this:PaymentError/>
        <exception ref=any/>
    </concurrent>
    <resolution>
        <exception ref=this:Abort/>
    </resolution>
</rule>
```

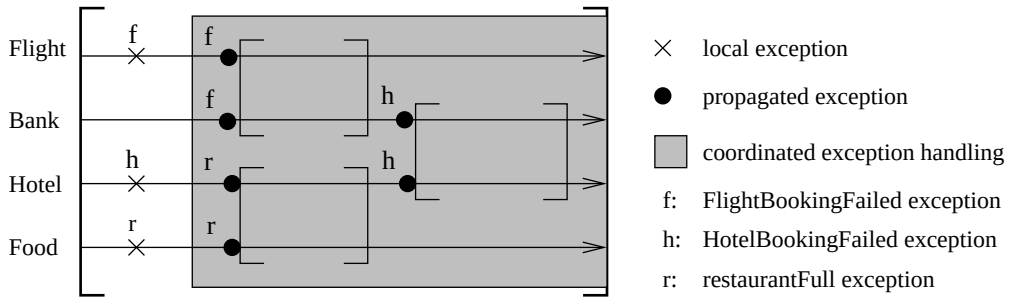


Figure IV.11: Concurrent exceptions in the travel agency WSCA

The figure IV.11 illustrates the execution of coordinated exception handlers when three exceptions occur concurrently. We have the *FlightBookingFailed* and *HotelBookingFailed* exceptions, for which rules are given in the previous document, and a third exception *restaurantFull*, which does not appear in the document, raised by a participant in charge to reserve a table in a restaurant close to the chosen hotel. As shown on the figure, the handling of the flight booking exception is performed before the one for hotel booking. The handler for the restaurant reservation failure can however be performed at the same time as the flight booking exception handler as it only involves the *Hotel* and *Food* participants.

IV.3 WSCAL orchestration language

This section details the language used to describe the standard and exceptional behaviour of WSCA operation participants and of *before*, *after*, *abort* and *failure* blocks of WSCA operations and of nested actions.

The behaviour of each participant is specified separately within the $<$ *participant* $>$ element (see Section IV.2.2). All participants start their execution synchronously, after the execution of the $<$ *before* $>$ block. The structure of the $<$ *participant* $>$ element is as follows:

```
<participant name=NCName>*
   <state>?
      <xsd:schema ... />*
   </state>
   <behaviour>
      <try>*
         Statements ...
         <coordinatedHandler exception=QName>*
            Statements ...
         </coordinatedHandler>
```

```
        </try>
    </behaviour>
</participant>
```

The < *state* > element contains declaration of local state variables used for local computations during the execution of the participant. These variables are only seen by the participant that declares them. The execution flow of the participant is further declared within the < *behaviour* > element. The behaviour of the participant subdivides into the participant's standard and exceptional behaviours. They are defined by the < *try* > and < *coordinatedHandler* > elements, which contain language statements for calling Web services, accessing local and shared variables, starting nested actions, synchronizing with other participants and conditional statements.

Each XML element has an optional *name* attribute, which can be used to identify the statement.

IV.3.1 Sequential execution

The < *sequence* > construct enables specifying a set of statements that are to be executed in lexical order.

The syntax of < *sequence* > is given by:

```
<sequence name=NCName ?>
    Statements ...
</sequence>
```

The < *sequence* > block terminates when the last statement terminates or if a non handled exception occurs within the current block.

IV.3.2 Parallel execution

The < *all* > statement specifies a set of statements that can be executed in any order or in parallel. The syntax of the < *all* > construct is given by:

```
<all name=NCName ?>
    Statements ...
</all>
```

The < *all* > block terminates when all embedded constructs terminate their execution, or if an exception is raised in a top-level context within the < *all* > element.

IV.3.3 Conditional execution

The < *switch* > statement executes one of the embedded < *case* > blocks, based on their condition expressions. Conditions are set as an XPath boolean expression for each < *case* > element with the *condition* attribute. < *case* > conditions are evaluated in the lexical order and the first one that evaluates to *true* is executed. Then, remaining < *case* > conditions are evaluated sequentially and executed if they evaluate to *true*. If no branch is selected, then the < *default* > block is executed, if any. Otherwise, the < *switch* > action terminates.

The syntax of the < *switch* > construct is as follows:

```
<switch>
  <case condition=Boolean-expression> +
     Statements ...
  </case>
  <default> ?
     Statements ...
  </default>
</switch>
```

IV.3.4 Iteration

The < *while* > element specifies a looping block that executes repeatedly as long as its condition evaluates to *true*. The condition is set as an XPath boolean expression, with the *condition* attribute. The condition is re-evaluated at each iteration at the beginning of the block. The < *while* > construct terminates, without executing the block, when the condition is evaluated to *false*.

The syntax of the < *while* > construct is as follows:

```
<while condition=Boolean-expression>
   Statements ...
</while>
```

IV.3.5 Interactions with composed Web services

The < *call* > statement allows specifying interactions with composed Web services. An interaction is realized by calling an operation offered by the Web service. The messaging behaviour of the interaction, i.e., the order of input and output messages, is defined by the Web service's WSDL interface that specifies the message exchange pattern that is required to be implemented. The < *call* > construct then includes declaration of message types such as input, output and fault messages, and interactions are performed by the run-time support system according to the messaging behaviour of the Web service operation. The same construct can thus be used for synchronous RPC-like request-response operations as well as for asynchronous one-way operations.

Additional properties related to each operation can be declared by a number of attributes: It can be specified if the invocation is retry-able or if an alternate composed Web service can be selected in case of unavailability of the service or in case of a failure of the initial interaction. A timeout can also be set for the termination of the call. Note that these attributes, when defined explicitly, override any default behaviour assumed by the underlying run-time system. If they are not set explicitly, a default value is set but this default value can be overrided by the run-time.

The call construct further includes statements for specifying the behaviour in case of abortion of the interaction that may be caused for example by a propagated exception and an action that should be executed if a timeout occurs.

The syntax of the < *call* > construct is as follows:

```
<call name=NCName ?
      service=QName
      operation=QName
      abortable=boolean:true
      retry=nonNegativeInteger:0
      tryAlternate=boolean:false
      timeout=Duration ?>
   <input message=NCName/>*
   <output message=NCName/>*
   <fault message=NCName/>*
   <onAbort> ?
      Statements ...
   </onAbort>
   <onTimeout> ?
      Statements ...
   </onTimeout>
</call>
```

The composed Web service to be invoked is specified using the *service* attribute and the associated operation with the *operation* attribute. Note that the service name must refer to one of the abstract services as defined in the $< services >$ element associated with the WSCA. Input, output and fault messages associated with the called operation are given with respective sub-elements that must refer to variables seen by the given participant.

The *abortable* attribute is used to declare if the invocation can be interrupted with an external signal, such as an external exception propagated to the participant leading the participant to terminate its execution. If set to *true*, the invocation is interrupted by immediately closing the connection, and the $< call >$ construct terminates after performing actions declared in the embedded optional $< onAbort >$ construct. Any messages that might be received from the composed Web service are discarded upon the abortion of the call.

The *retry* attribute specifies how many times the call can be repeated if the interaction fails. It does not specify any compensation action to be performed by the participant before the retry; the composed Web service operation is assumed to be retry-able. If an application-specific retry strategy is to be implemented, the *retry* attribute should not be set and the Web service call must be invoked within a loop construct, with appropriate exception handlers for performing compensation operations.

The *tryAlternate* attribute specifies whether an equivalent alternate Web service could be invoked or not in case of failure or unavailability of the initial Web service. Recall that multiple Web services can be bound for a single abstract Web service in the $< services >$ elements within the declaration of the WSCA composite service, either by giving a static list of alternative Web services or relying on a dynamic binding support. However, multiple binding only applies to the selection of the Web service at the first invocation and subsequent interactions with the same abstract Web service are always performed with the same Web service. Setting the *tryAlternate* attribute to *true* allows invoking the operation on a different Web service even if the current Web service has already been invoked by the current participant or by another. When such an alternate Web service is bound, the WSCA keeps the new binding and discards the previous one. Other participants are also affected by the new binding, except for ongoing interactions that are allowed to terminate with the same Web service.

A timeout can be set with the *timeout* attribute for each operation. The timeout specifies that a timeout event will be triggered if the Web service operation does not return a response (either an output or a fault message) until expiration of the specified duration. The timeout applies to all call attempts, including alternate

Web services and retries, i.e., the timer is not re-initialized at each call. When the timeout occurs, the interaction is interrupted by terminating the connection to the Web service and the $< onTimeout >$ block is executed. Any incoming message after that the timeout event is triggered from the composed Web service is discarded.

IV.3.6 Assign

The $< assign >$ statement is used to change the value of a variable to a new value. Any variable that is visible in the current context of the participant can be assigned a new value, such as a participant's local variable or a WSCA's shared variable. The new value can be given using an existent variable, by an element defined in the current context, or it can be computed using XPath expressions.

The general form of an assignment is as follows:

```
<assign name=NCName ?>
   <to var=NCName />
   <from
      var=NCName ?
      expr=XPath-expression ?>
      value
   </from>
</assign>
```

The $< to >$ element specifies the destination variable and the $< from >$ element defines the source value. The new value may be given by assigning the value of an element referenced using the *var* attribute, by an XPath expression using the *expr* attribute, or by directly specifying the value in the form of XML elements.

IV.3.7 Empty

The $< empty >$ constructs is an operation that does nothing. It can be used when an operation is needed but no action is to be performed, for example for setting a synchronizing point for a participant.

The syntax of the $< empty >$ construct is given by:

```
<empty name=NCName ?>
</empty>
```

IV.3.8 Waiting

The < *wait* > construct allows waiting for a period of time or until a deadline is reached. Waiting is immediately aborted when an exception occurs.

We get:

```
<wait name=NCName ?
      type=duration|absoluteTime:duration
      timer=Duration >
</wait>
```

The *type* attribute specifies whether if the timer defines a time period (*duration*) or for an absolute time (*absoluteTime*).

IV.3.9 Synchronizing participants

The < *join* > statement is used to synchronize two or more participants belonging to the same WSCA operation, or to the same nested WSCA. When a participant executes a < *join* >, the execution of the participant is suspended until the statement referenced by the < *join* > statement is executed by another participant, or in a parallel process of the same participant. Note that participants involved in a nested WSCA cannot synchronize with participants outside the nested WSCA, nor with participants of sibling nested WSCAs. In addition a timeout can be set on the waiting time to allow continuation of the participant when the action is not terminated until the timeout.

We get the following syntax:

```
<join name=NCName ?>
   <condition action=QName >+
   <timeout type=duration|absoluteTime:duration
            timer=Duration > ?
   <onTimeout> ?
      Statements ...
   </onTimeout>
</join>
```

The < *condition* > element provides the name of the statement whose termination is waited for. Multiple conditions may be specified by stating as many < *condition* > elements as necessary. The join condition is satisfied when all actions terminate, regardless of the lexical order, and the < *join* > action terminates resuming the execution of the participant. The < *join* > can be inter-

rupted prematurely by an exception or if a timeout occurs. The $< timeout >$ element enables setting a timeout for the waiting time and has a similar syntax to the $< wait >$ statement. In addition, a timeout occurs if the statement waited for can no longer be executed, for example if the block construct or the participant containing the specified statement terminated without executing the statement. When a timeout occurs, the $< onTimeout >$ block is executed, if any. The $< onTimeout >$ block can in particular be used to raise an exception.

IV.3.10 Throwing exceptions

The $< throw >$ statement raises an exception whose handling is specified using $< try > ... < catch >$ constructs for defining exception handling scopes.

The syntax of the $< throw >$ construct is as follows:

```
<throw name=NCName ?
       exception=QName
       exceptionData=NCName ?>
   Value ...
</throw>
```

The name of the exception being raised is specified using the *exception* attribute. The exception must be globally unique and is referenced using a QName. The *exceptionData* refers to a variable that might be used to define additional informations that can be passed to the exception handler context, or to enclosing scopes by propagation if a immediate handler context is not present. The exceptional data value can also be defined statically in the content of the $< throw >$ element.

IV.3.11 Exception handling scopes

Exception contexts and associated exception handlers are declared using the $< try > ... < catche >$ and $< try > ... < coordinatedHandler >$ statements for respectively local and coordinated exception handling:

```
<try name=NCName ?>
   <try name=NCName ?>
      Statements ...
      <catch exception=QName ?
             exceptionData=NCName ?> *
         Statements ...
```

```
        </catch>
    </try>
    Statements ...
    <coordinatedHandler exception=QName ?
                        exceptionData=NCName ?> *
        Statements ...
    </coordinatedHandler>
</try>
```

The $< try > ... < catch >$ blocks define local exception handling scopes, which can also be nested. Any exception that is raised within a $< try >$ block causes the execution of the block to be terminated. Then, if a corresponding exception handler is defined, it is selected and executed. The $< catch >$ element is used to specify the exception handler. The exception that is handled is referenced by the optional *exception* attribute. If no exception attribute is defined, the $< catch >$ block acts as a default exception handler for all exceptions. After that, the execution of the participant continues at the end of the $< try >$ block. Otherwise, if an unhandled exception occurs, the exception is propagated to the enclosing scope. If the propagated exception reaches the top-level $< try >$ block, the exception is to be propagated to all the action's participants for coordinated handling. Handlers for coordinated exception handling are further defined with the $< coordinatedHandler >$ elements. Note that more than one coordinated exception handler for different exceptions can be executed in case of concurrently raised exceptions, according to the concurrent exception resolution based on the rules defined for the WSCA operation or nested WSCA (see Section IV.2.5).

IV.3.12 Starting a nested WSCA

The $< startNested >$ statement causes the participant to enter a previously defined nested WSCA with a $< nestedWSCA >$ element (see Section IV.2.3). A $< startNested >$ element contains declarations for specifying the actions to perform by the participant within the nested action. Its structure is thus similar to the one of a WSCA operation participant. We get:

```
<startNested action=QName>
    <state>?
        <xsd:schema ... />*
    </state>
    <behaviour>
        <try>*
            Statements ...
            <coordinatedHandler exception=QName>*
                Statements ...
```

```
            </coordinatedHandler>
        </try>
    </behaviour>
</startNested>
```

IV.4 Concluding remarks

This chapter has described the Web service composition action language for the dependable composition of Web services.

We proposed in this chapter a model for building dependable composite Web services and presented the Web Service Composition Action declarative language. A WSCAL document describes a WSCA composite Web service by specifying all the operations provided by the composite Web service and the concurrency support of a WSCA instance. A WSCA operation is further specified as a concurrent program executing several participants, each of them interacting with one or more composed Web services. Coordination of participants is ensured through nested actions, which provides isolated accesses to composed Web services and nested transactional accesses to shared variables. The recovery strategies that can be specified are based on a coordinated exception handling model, where several participants co-operatively execute exception handlers.

The Web service composition action language has a number of features that are novel in Web service orchestration languages and coordination protocols. Compared to existing coordination and transaction protocols for Web services, which require from Web services to support the given protocol to be part of the composition, the key advantage of WSCA is its ability to define recovery actions involving several Web services, which are not necessarily dependable. Another particular interest is the abstraction of bindings with composed Web services, which allow to discover and bind composed Web services at deploy-time or dynamically at runtime. The benefit for the composite Web service is an enhanced availability and reliability in accessing composed Web services.

Moreover, declaration of required conversations from composed Web services is very useful. The conversation can in particular be used for the discovery of matching Web services and for verifying the correctness of interactions at runtime. The next chapter proposes such tools based on the exploitation of the specification of conversation supports of Web services with the WS-RESC language (see Chapter III). A runtime support for executing WSCAs is further presented and assessed.

V Performance and Experiments

This chapter presents some experimental results which both motivate and validate building composite Web services using WSCAL and WS-RESC. Composite services are developed using the WSCAL language and the WS-RESC conversation language is used to declare required conversations of composed Web services. First, the development process of WSCA composite Web services is presented. Integration with WS-RESC is presented in the following section with performance measurements for the service discovery protocol and for the on the fly verification of invocation correctness. Then, the WSCA runtime is presented and assessed for performance and reliability, by comparison with related solutions in the area.

V.1 WSCA development

A general view of the WSCA composite Web service development, deployment and execution was introduced in the previous chapter (see Figure IV.1, page 78). Programming a WSCA composite Web service comes down to three steps:

(1) Providing the required interface and configuration information for the WSCA composite service. The programmer describes abstract services corresponding to composed Web services and the concurrency behaviour of the composite service application. This is done in the *services* part of the WSCA document. The necessary stubs for calling the discovered composed Web services are generated at deploy time.

(2) Programming the application logic of each operation of the composite Web service. Application logic of the WSCA service is written in the WSCAL orchestration language. Developing with WSCAL abstracts many programming details such transactional accesses to local resources, SOAP calls to

composed Web services and dynamic binding mechanisms. The resulting composite Web service is intended to be executed on a Web service application server. This is similar to developing Web services in BPEL [BEA Systems et al., 2005] but it is more efficient because of the structuration that imposes WSCAL for building dependable applications and the built-in dynamic binding mechanism based on service discovery protocol.

(3) Deploying the WSCA service on a Web service application server. The application server can be a dedicated WSCA server, which can interpret the WSCAL language. Alternatively, the WSCAL declaration can be compiled into an executable Web service application to be deployed on a Web service application server. In our prototype implementation we chose the second option for efficiency. We implemented generic Java classes for the WSCA runtime support and the WSCAL description is transformed to Java code using these classes. The resulting application is then compiled and deployed on an Apache Tomcat application server and makes use of the AXIS SOAP engine for interacting with service requesters and composed Web services. Additionally, the WSDL document of the WSCA composite Web service is to be provided for describing the composite service. Clients that download the WSDL document can invoke WSCA operations of the composite Web service.

A complete example for the travel agency composite Web service can be found in Appendix C. This example is used in Section V.4 for assessing performance and reliability of the composite service.

Composed Web services are selected through a service discovery protocol based on the matching of abstract Web service descriptions and on the matching of conversations specified using WS-RESC. The next section presents an implementation of such a discovery protocol using a UDDI registry. The performance of matching algorithms are measured because efficiency is a key parameter for considering service discovery at runtime with dynamic binding.

V.2 Service discovery

Composed Web services are described abstractly in the $< services >$ element of the WSCAL description (see Section IV.1.3). For each composed Web service two documents are given. The first document is the abstract part of the WSDL document and the second document defines the conversations in WS-RESC that are required to be supported by the composed Web service.

V.2.1 Matching abstract WSDL descriptions

The abstract declaration of composed Web services comprises type information, exchanged messages and operations. In WSDL, these correspond respectively to the $< type >$, $< message >$ and $< portType >$ elements. For the WSDL matching, a required abstract Web service matches a service description if all the operations defined in the abstract declarations are offered by the Web service. Corresponding operations should define the same set of messages, with the same typed parameters. Note that the Web service can include more operations than the required service. This does not affect the matching as these operations will not be used in the composition.

We considered using a UDDI registry because it provides a standardized API for both searching and registering Web services. The first step is to retrieve WSDL interface descriptions of Web services that are to be potentially integrated. Composed Web services are registered in a UDDI registry by their respective providers. The service's abstract interface, which is constructed using the information from the WSDL service interface description is published by the service provider in a UDDI registry as as a tModel, which is technical specification. Typically, for Web services, the technical specification contains a URI pointing to the actual WSDL specification (using the *overviewURL* field). At client side (the WSCA), we use the UDDI find_tModel message to find tModels of WSDL service descriptions. This message will return a list of tModel keys. Interface descriptions are then retrieved using the UDDI get_tModelDetail message. Each interface description has a URI referencing the exact location of the WSDL document, which is also retrieved. At the end of this process we have a list of WSDL interface descriptions. The next step is then to search among these WSDL interface definitions those that include all the operations and associated typed messages defined in the required WSDL document. A more efficient approach would be to retrieve only partially or fully matching description from a UDDI server by querying the registry with the required WSDL document, but the UDDI API does not define such a retrieve method.

The implementation is straightforward. First we check if all operations defined in the required WSDL are present in the provided WSDL by syntactic matching, in the same *portType* element, which defines a set of operations. If at least one operation is missing, the check fails. If all operations are present, then we associate input messages of the required operation with input messages of the provided operations, and the same for output and fault messages. If one association fails, i.e., if an operation does not define a required message or if it defines a message not defined in the required WSDL, the check fails. If all associations are correct,

we check the type compatibility of message parts. If all tests succeeds, the check succeeds and the provided WSDL is marked as matched.

We run the WSDL matching algorithm implemented as a Java application, on a set of 1016 Web service descriptions retrieved from main public UDDI registries[1] using the UDDI4J Java class library[2], which provides an API for interacting with UDDI registries. The sizes of the WSDL documents that are retrieved vary from 1.4 KB to 190 KB, and they each define from 1 to 58 operations. It is worth noting that the number of operations of Web services are generally low. For the retrieved Web services, the average number of operations was 5 and the median number of operations is only 3. The average WSDL document size is about 15.4 KB. The experiment has been performed on a 2.6 GHz Pentium IV, Linux machine with 512 MB of RAM and IDE hard disks. Queries to UDDI servers are sent over a broadband Internet connection.

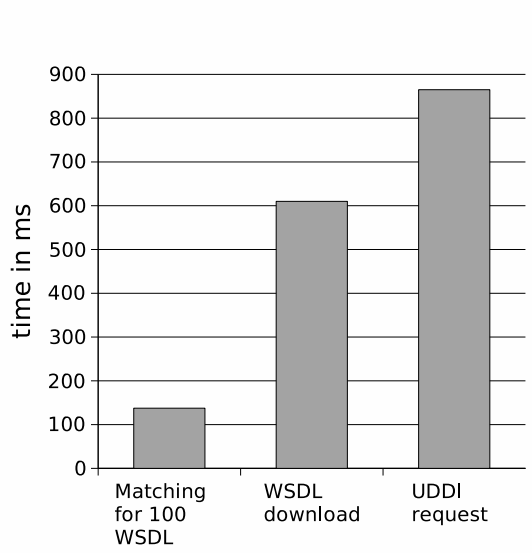


Figure V.1: WSDL matching

Figure V.1 gives the average time for retrieving one WSDL document and for querying a UDDI registry and the total time for matching 100 WSDL documents. The average execution time of the WSDL syntactic check over these files are about 1.375 ms per WSDL document, which is negligible compared to the average WSDL retrieval time from different service providers, which was measured

[1]The UDDI Business Registry, which operates as a distributed service is used. The nodes provide replicated data and can be accessed at the following addresses: SAP UDDI Business Registry at `http://uddi.sap.com/`, IBM UDDI Business Registry at `http://uddi.ibm.com`, Microsoft UDDI Business Registry at `http://uddi.microsoft.com`

[2]UDDI4J is available with a free software license at http://uddi4j.sourceforge.net/

to be 600 ms in average per WSDL document if the service is available. Note that retrieving WSDL documents can be parallelized. Furthermore, retrieving tModels containing the references to WSDL documents from a UDDI registry using a single UDDI query took in average 850 ms using different registries at different hours. The average total time is relatively short but it can be problematic for dynamic binding with service discovery performed at run time. However, this can be drastically shortened using local caches of retrieved WSDL documents, and even more by caching matching results. In the former case, only the syntactic matching of required service definition with the list of WSDL documents would be performed, with an average time of 1.375 ms per WSDL document, which is short enough to be considered for runtime service discovery. Furthermore, the number of WSDL documents to retrieve is restricted by a pre-selection. For example, in the case of a flight reservation Web service, only Web services of transport companies and travel agencies may be retrieved.

The process described above returns a list of service interfaces, whose WSDL specifications match the required service interface. If there is no more requirements on the services, such as specific conversation supports, Web service instances implementing these WSDL documents can then be retrieved by requerying the UDDI registry, using the matching WSDL documents' references (tModelKeys). The binding templates returned from the UDDI registry will then include access points to particular implementations of the WSDL documents. However, if the WSCA composite service defines in addition to the abstract WSDL definition, a list of conversations that are required to be implemented by the composed Web services, we should also match required and provided conversations.

V.2.2 Conversation compatibility checking

A Web service conversation *matches* a required conversation if for all possible execution of the required conversation there exists at least one corresponding execution path among the conversations supported by the Web service. In other words, if we represent the required conversation with a process graph P, the service conversation process should be able to *simulate* P. We implemented a simulation relation verification algorithm over conversations of the required and provided services specified in WS-RESC.

A simulation pre-order is a relation between state transition systems associating systems which behave in the same way in the sense that one system simulates the other. Formally,

Given a labelled transition system (S,$\mathcal{A}$,$\rightarrow$), where S is a set of states, $\mathcal{A}$ is a set of labels and $\rightarrow \subseteq S \times S$ is a binary relation over S of transitions, a binary relation $R \subseteq S \times S$ is a simulation if whenever $(p,q) \in R$ then for each $a \in \mathcal{A}$,

$$\text{if } p \xrightarrow{a} p' \text{ , then } q \xrightarrow{a} q' \text{ such that } (p',q') \in R.$$

A process p is simulated by a process q if there is a simulation R such that $(p,q) \in R$.

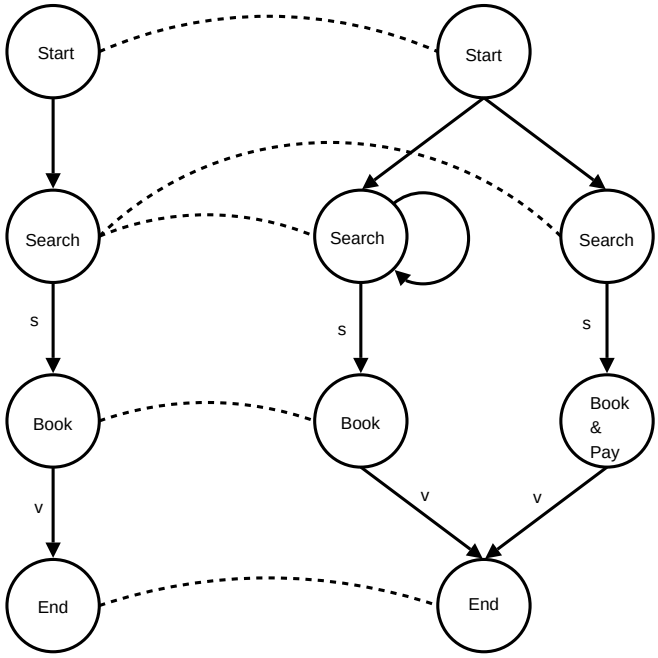


Figure V.2: Simulation relation

Figure V.2 illustrates two conversations, where the simulation on the right-side simulates the conversation on the left-side. The simulation relation is verified by associating each move of the conversation on the left, to one or more moves of the conversation on the right.

The WS-RESC descriptions of both required and provided conversations are parsed and stored in an internal data type for computations. The Xerces 2 Java XML parser [3] and the JGraphT Java graph library [4] have been used respectively for parsing XML files and representing graphs.

[3]Apache Xerces2 Java XML parser is freely available at `http://xml.apache.org/xerces2-j/`

[4]JGraphT is freely available at `http://jgrapht.sourceforge.net/`

The simulation relation verification algorithm follows the following steps. If the check fails at any of these steps, the verification stops, and the provided conversation is considered as not matching. The first three steps are introduced for increasing performance of the verification process by only verifying the presence of some required states.

(1) The initial *Start* states of conversations and their output transitions are verified. For each start transition of the required conversation, there must exist a start transition in the provided conversation. If one start transition is not present, the algorithm stops and matching fails.

(2) The final *end* states of conversations and their input transitions are verified. Similarly to the above step, all final *End* states of the required conversation and their related input transitions are verified.

(3) If all the previous checks succeed, presence of all intermediate states of the required conversation in the provided conversation is verified.

(4) Starting from the start state, and for each state of the required conversation, the output transitions are checked and a relation is established between states of the required conversation and the provided conversation. The check is performed recursively for each state of the required conversation with the related states of the provided service. If output transitions of all states defined in the required conversation match, we can deduce that to all moves of the required conversation there is a matching move at the provided service.

Transitions are checked by comparing the Web service operations that are referenced in the corresponding destination states and by verifying the equality of labels, which correspond to messages. Recall that the absence of a label means that there is no condition on the transition, and any message can be emitted by the operation (see Section III.1.1). Thus, if no label is present at the required conversation's transition, then no label must be present at the provided conversation's transition. However, the absence of a label at the provided side causes the transition to match any label defined in the required conversation. Furthermore, time constrained transitions match if the time constraint on the transition of the required conversation is equal to or shorter than the related one of the provided conversation.

We tested the efficiency of the simulation relation verification with randomly generated required and provided service definitions, using the WS-RESC conversation description language. We assume that the WSDL descriptions already

match, i.e., that all required operations are present in the provided services and their message types match. However, the provided services can include additional operations leading to more complex conversations. For each operation, we defined two different messages representing a normal output message and a fault message.

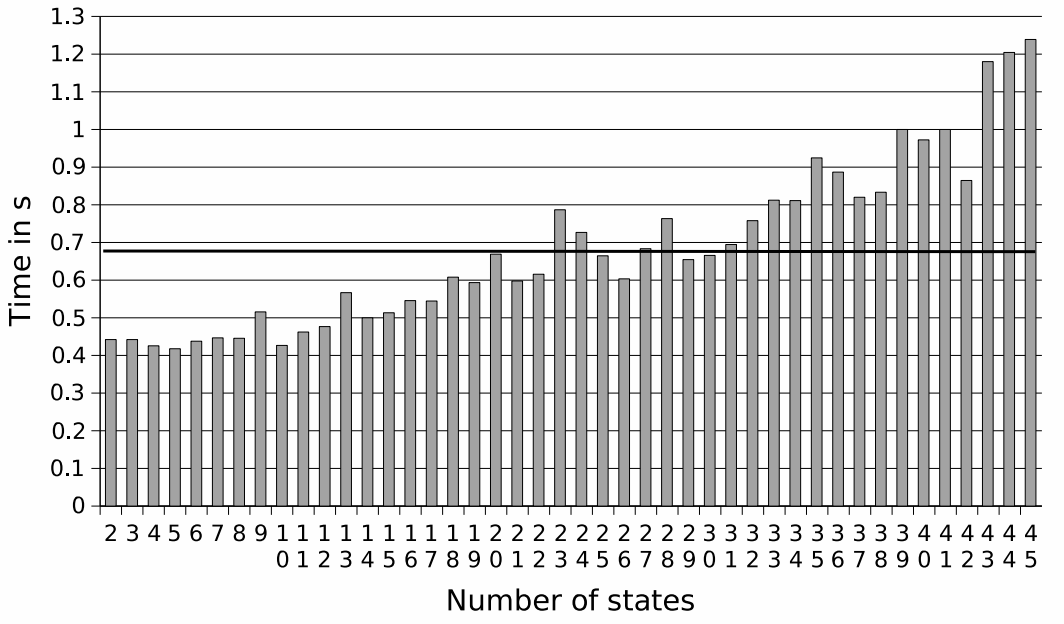


Figure V.3: WS-RESC matching

Figure V.3 presents tests results with varying numbers of states of the required conversation. Since our algorithm terminates the verification as soon as a required state can not be related to a provided state, we only take into account in the measurements the worst case where the two conversations match, and all steps should then be performed up to completion. For this, first is constructed the required conversation, and then a set of provided services are constructed by adding states and transitions to the required conversation without breaking compatibility. As a result, the provided conversation always matches the required conversation, but is more complex with more states, transitions and different labels.

The average execution time of the verification algorithm was measured as 690 ms per WS-RESC document. This is similar to the WSDL document fetching time, but much higher than the syntactic match on WSDL files. The total time will vary according to the number of documents to verify. Note that these results are for matching services. For services that do not match, which can be the

case for the majority of services, the execution time is much lower. The lower bound is the system call execution time for opening the WS-RESC file, parsing XML and building the conversation graph, which is measured as 340 ms. WS-RESC matching increase the time for selecting a Web service, but it should be considered in the trade off analysis for increasing reliability.

If a WSCA defines required conversations for all or some of the composed Web services, it is the WSCA developer's responsibility to realize a composition flow that respects the conversations for all Web services involved in the composition. The next section deals with the verification of invocation at run-time.

V.3 On the fly verification of invocation correctness

Different analysis can be performed on the composition process to verify if it matches the required conversations (and hence, the provided conversations) of composed Web services. Programmers may be assisted with formal verification tools for either statically checking specifications or generating correct codes from specifications, given a formal encoding of both the behavioral specification and those of existing conversation descriptions. However, even in the case of a correctly verified behavior specification, errors may occur at run-time, such as time-outs due to deadlocks or unavailability errors due to network problems. Moreover, the composed Web service may not behave as it is advertised on its interface. In all cases, errors as well as behavioral mismatches should be detected and reported as an exception by the underlying run-time system to the participant that calls the composed Web service's operation. For run-time analysis, the actual conversation of the Web service instance being bound should be retrieved and used for verification as it might contain additional constraints not explicitly specified by the WSCA developer in the required conversation.

A client-side generic runtime verifier is implemented for checking conformity of Web service invocations issued by a service requester with individual conversation descriptions of Web services given in the WS-RESC language. Verification is made online at each interaction by a specialized component that intercepts incoming and outgoing messages (see Figure V.4).

Checks are performed at two stage, before and after each invocation for checking input and output conditions. Service request calls are intercepted first and checked if they match operations in a list of expected operations that is constituted at run-time and continuously updated, according to the states defined in the WS-RESC specification and the executed operations. If the outgoing mes-

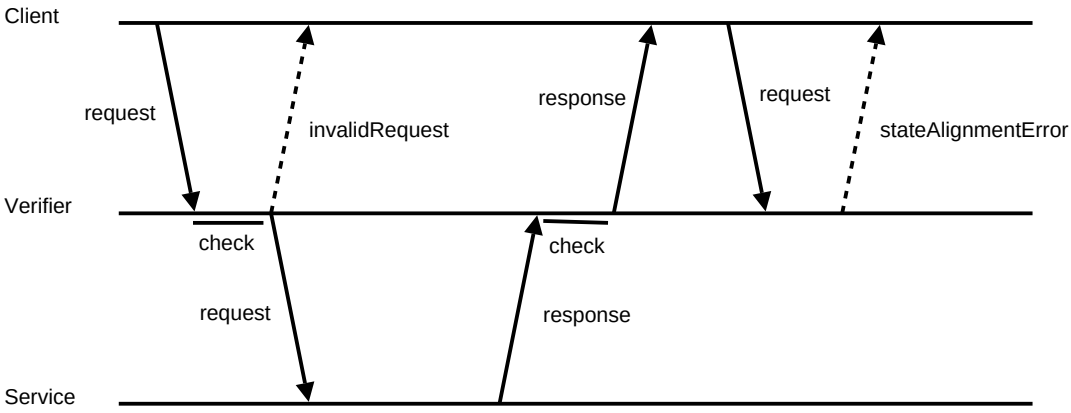


Figure V.4: Conversation verifier

sage contains an expected operation, the message is forwarded as is to the Web service. Otherwise, a pre-defined system *invalidRequest* is signaled to the service requester with additional information on the exception such as the reason of the fault. The Web service is never invoked and the message is discarded. In the second stage, the response message sent by the Web service is intercepted and analyzed. Depending on the output message name and the transitions defined in the WS-RESC description, the list of expected operation names for this Web service is updated and the message is forwarded as is to the requester. If the return message does not correspond to any of the expected transition conditions as defined in the WS-RESC definition, the return message is not discarded and sent to the service requester. However, the state of the conversation for this Web service is marked as *unknown* by the verifier, causing the rejection by the verifier of all subsequent requests to this Web services and the signaling of *stateAligne-mentError* exception to the service requester.

Figure V.5 gives the results of experiments done using automatically generated clients and services. The clients' invocations are constructed so that the conversations are respected, to measure the worst cases. For each interaction, first is measured the total execution time without the on the fly verification and then, with the verification. The average execution time of a Web service invocation without on the fly verification is measured as 240 ms, with the service and the client deployed on different networks, connected over a broadband Internet connection. The overhead of the on the fly verification varies according to the complexity of the conversation description. Without considering the initial setup time for reading the conversation and parsing it, the average overhead of the verification by operation is measured about 90 ms for a set of conversations of 2 to 45 states. Compared to the average Web service invocation time for one opera-

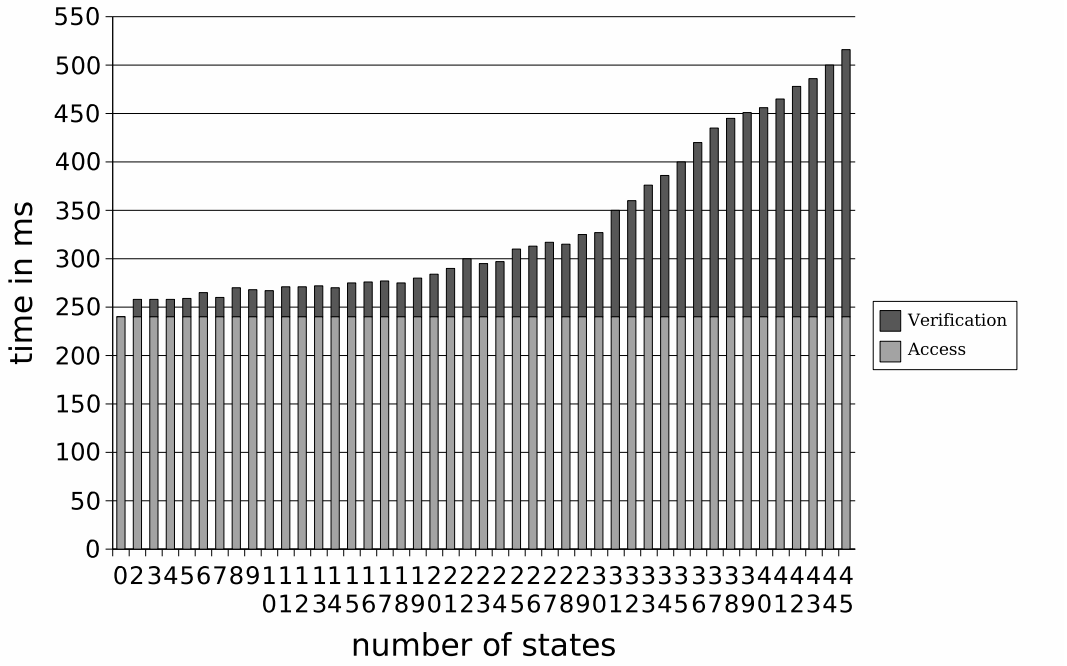


Figure V.5: Verification cost

tion (240 ms), the extra time needed for verifying correctness is short enough, and can be considered to include it in most Web service interactions.

We integrated an on the fly verification component in the WSCA runtime. But, verification of conversation matching is performed only if strict conformity with a given conversation definition is required at the WSCA design time by setting the value of the *strict* attribute to *true* in the $< services >$ element of the WSCAL definition. Additionally, the conversation document that is used to do verifications is the one that is specified by the WSCA designer and referenced with the *conversation* attribute in the same declaration. A further improvement would be to retrieve dynamically the provided conversation definition of Web services, which can be included as an extension in the WSDL definition or referenced in the UDDI registry. However, a standardized specification does not exist yet neither for describing conversations, nor for attaching it to the service definition.

V.4 WSCA runtime

A WSCA composite service is built by transforming the WSCAL specification of the WSCA into Java code, using a set of Java classes implementing the base WSCA principles. Participants are implemented with threads in a main class, and a controller component is used to synchronize them in nested actions and to propagate exceptions. A concurrency control manager further controls accesses to composed Web services by allowing or delaying accesses. A dynamic binding mechanism allows localizing services on the fly and integrating them in the composition in case of unavailability of an already bound service, detected by the occurrence of time-outs. The exception handling mechanism includes the detection of faults, mechanisms for propagating exceptions among participants, resolving concurrent exceptions and synchronizing participants for realizing coordinated exception handling. The internal exception mechanism of the Java programming language is used for local exception handling. Propagation of exceptions among participants is coordinated using a controller object. Participants invoke a synchronized call on the controller to check the occurrence of a propagated exception and for signalling an exception to propagate. Checks are performed before entering a nested action, before calling a composed Web service and before terminating.

The next section presents the WSCA runtime using an example application and compares the execution performance of a composite Web service built using WSCAL and an equivalent BPEL based composite service. Then, the implementation of the concurrency control is presented by analyzing performance increase by relaxing isolation based on the WS-RESC description. We finally assess the reliability of a WSCA composite Web service by comparing with implementations of existent specifications for dependable Web service compositions.

V.4.1 Comparing WSCA design and execution

We use build the composition workflow of the travel agency composite Web service using WSCAL and BPEL, which is the most mature composition language available for which several execution engines have been developed. The objective is to compare the overall execution performance. Dependability related measurements are further given in the next section.

The travel agency composite Web service consists of a flight reservation subsystem and of a hotel reservation subsystem. The flight reservation subsystem

make requests to a set of flight booking Web services for finding a flight to a given destination. The hotel subsystem similarly accesses a set of accommodation reservation Web services to book an hotel room for the duration of the trip. If a flight or hotel room became unavailable during the search phase, then alternative Web services should be contacted to complete the reservation. If no other options can be found, the trip is aborted by notifying the requester that no trip can be found. The booking phase should be initiated only when all searches are completed. If during the booking, one of the booking fails, the other booking should be stopped. If it is already performed, it should be cancelled. Furthermore, Figure V.6 represents the standard execution of a WSCA composite Web service operation, composed of two participants *Flight* and *Hotel*, entering two nested actions *Search* and *Book*, and accessing to composed Web services *Flight reservation Web service* and *Accommodation reservation Web service*.

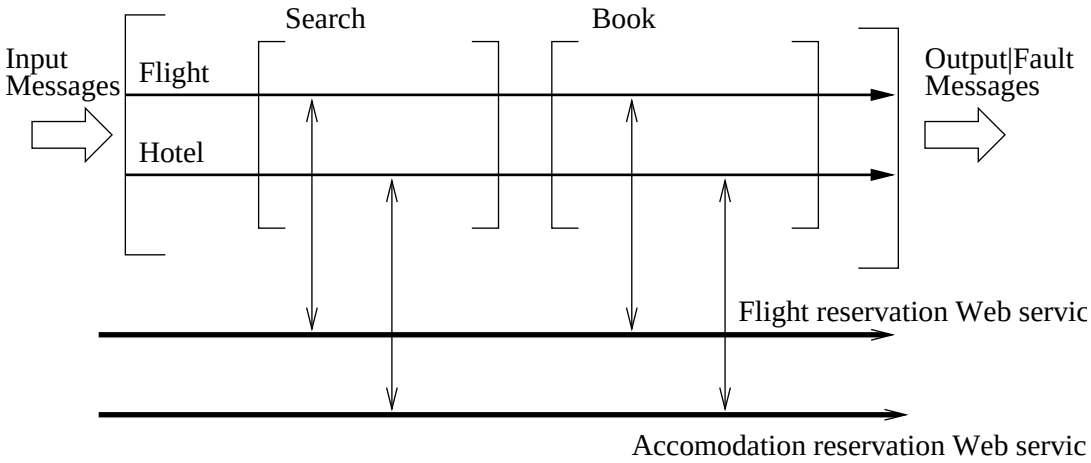


Figure V.6: Travel agency WSCA execution

The WSCA version is specified without considering dynamic binding, abstract service declaration and WS-RESC verification, as these features are not supported in BPEL. The WSCAL specification is used to generate a Java code, which is compiled with associated WSCA runtime classes and deployed on top of the Apache Tomcat application server version 5 [5] and using the Apache AXIS SOAP engine version 1.2.1 [6]. The BPEL version is written according to the BPEL4WS specification v1.1 and executed on top of the open source BPEL engine ActiveBPEL, which is implemented in Java and executes on top of the Apache Tomcat application server version 5. Contrary to our implementation, ActiveBPEL does not compile the BPEL specification into an internal format

[5] `http://jakarta.apache.org/tomcat/`
[6] `http://ws.apache.org/axis/`

but interprets a process definition created from the BPEL specification at deploy time. Furthermore, the implementation uses also the AXIS SOAP engine to interact with composed Web services. There exists other implementations of BPEL engines, such as the Oracle BPEL Process Manager [7], Microsoft's BizTalk server [8] and Parasoft's BPEL Maestro [9]. We used the ActiveBPEL engine because it presents similar characteristics as the WSCA runtime (they are both written in Java and use Apache Tomcat and Apache Axis) and because it was freely available.

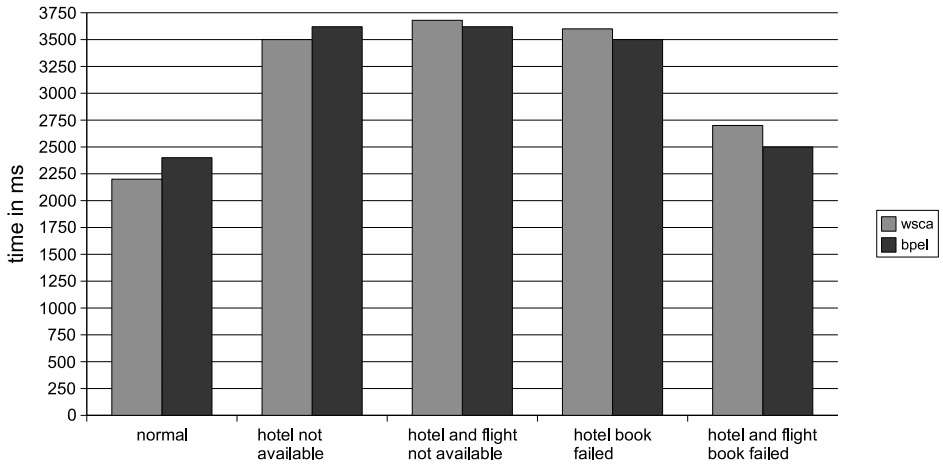


Figure V.7: WSCA vs BPEL

Figure V.7 gives total execution time for the two composite Web services, invoked with the same requests. Different cases are considered for both normal execution without any error and exceptional executions where flights and hotels are not available and payments fail. Tests are realized using the same service requester client, written in Java and using the AXIS SOAP container. Each test is executed 1000 times to compute the average execution time. The overall execution times are quite similar, which is mostly due to the fact that the major cause of delays are the access time needed to interact with Web services using XML based messages. When the composed Web services are deployed on a LAN for reducing the access time, we measure a better performance for WSCA, which does not interpret XML codes.

Another test is realized in a highly stressed environment by invoking the composite Web service simultaneously by multiple service requesters (see Figure V.8).

[7] http://www.oracle.com/technology/products/ias/bpel/
[8] http://www.microsoft.com/biztalk/
[9] http://www.parasoft.com/jsp/products/home.jsp?product=BPEL

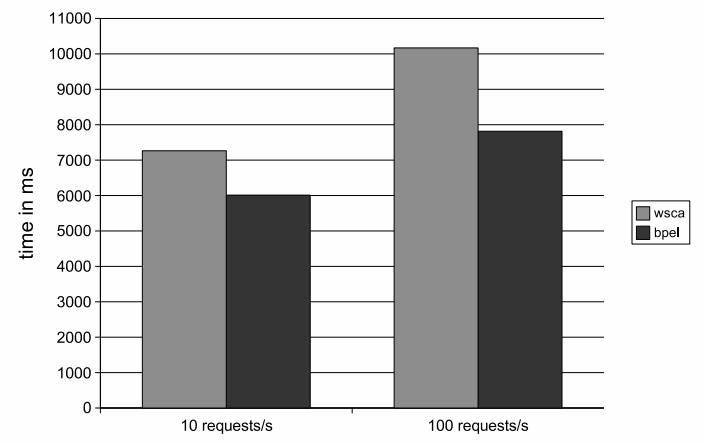


Figure V.8: WSCA vs BPEL in highly stressed environment

The average waiting time for WSCA clients is slightly longer, mostly due to local locking. However, all requests complete with no deadlocks or timeouts. The next section details the implementation of the concurrency control in the WSCA runtime and shows how and under which conditions isolation can be relaxed to increase performance.

V.4.2 Concurrency control

The main role of the WSCA concurrency management is to guarantee the expected isolation level of WSCAs when interacting with Web services by controlling concurrent accesses to them. The implemented concurrency control manager uses a lock based protocol for controlling accesses to composed Web services within a WSCA instance. The concurrency control includes a customizable conflict table, constructed by analyzing the WS-RESC description of composed Web services for relaxing isolation and increasing performance.

When a top-level WSCA starts, locks are acquired automatically on local objects associated to each Web service that is accessed during the WSCA execution. The locks give the access right to the participants that are involved in the WSCA to invoke any operation on the associated Web services. When a nested WSCA is started, the lock passes to the nested WSCA, and only the nested WSCA participants can access the composed Web services. All other accesses from participants outside the nested WSCA are delayed. Furthermore, all sibling nested action creations willing to acquire a lock for the same composed Web services are delayed as well. Lock are released at the and of nested actions and at the

end of the top-level WSCA.

The local lock based solution for ensuring isolation of concurrently running WSCA respect to composed Web services, raises several issues. First, is to decide when to acquire and release locks. A lock may be acquired at the initiation phase of the WSCA, or at the first access to a composed Web service. The second approach is chosen because it offers better performances with increased concurrency by shortening the lock holding time. This means that we allow nested WSCA creations even if the nested WSCA will access to an already locked Web service by another sibling nested action. Release of the lock should be done at the end of the WSCA holding the lock, once the WSCA terminates its execution. If a nested WSCA ends exceptionally, the lock is released, but the priority for acquiring a new lock is given to potential exception handlers for exceptions raised in parent actions.

For reducing the delay imposed by the design decision based on locks, the better approach is to assign the accesses to a particular Web service to a single participant. Furthermore, there are many Web services that support concurrent accesses by providing means for not interfering their results. For example, a Web service can used HTTP session cookies for identifying different, not interfering sessions. This capability is advertised by the Web service in the WS-RESC description. Based on this information, we can relax isolation to allow concurrency, and hence improve the overall performance by shortening the total execution time. The implementation of the concurrency control manager takes into account the WS-RESC description supported by Web services, which is analyzed at deploy time. Conversations that support sessions are identified, based on the *correlation* attribute. Then, a new local object is created at run-time whenever a new session is created by the participant that access the composed Web service. When a new nested WSCA accessing the same composed Web service is created, or when a new instance of a WSCA operation is invoked, the concurrency control manager checks the correlation value of messages to identify new sessions, which are executed without delay. Accesses using the same correlation value are not permitted and delayed until the lock on the local object associated to this session is released.

Another concurrency issue is related to concurrent accesses to composed Web services within a WSCA, from participants that all have the access right to call the Web service. By default, for increased performance, we allow concurrent calls from participants without restriction. Which means that if a Web service operation is invoked by a participant, another participant can invoke the same Web service even if the first operation has not terminated. This default behaviour can be customized to restrict concurrent accesses in case of potential conflicts

between operations. The concurrency control manager maintains a conflict table that is used to check if two Web service operations are conflicting or not. Generally, two operations conflict if the result depends on their execution order. We therefore implemented the conflict table in the concurrency control component that is customized according to the WS-RESC description of the Web service, if any. The conflict table gives for pairs of operations if they are conflicting by analyzing dependencies between operations and equivalence relationships. Two operations conflict if they are associated to two states of the same *activity* with a transition (specifying the ordering) between them. Furthermore, operations do not conflict if there is an equivalence relationship between activities where the two operations appears in different sequences. Construction of the conflict table is pretty fast, measured to be from 2 to 300 ms, depending on the complexity of the conversation, which is fast enough to include its construction at run-time for Web services discovered at run-time. However, the WS-RESC document retrieval time should be added though it can already be present if it was needed for the on the fly verification, conversation matching or if it is cached previously.

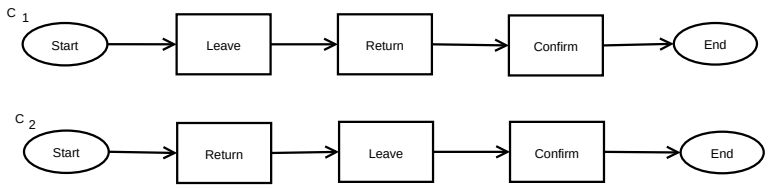


Figure V.9: Detecting conflicts

Figure V.9 represents two conversations C_1 and C_2 for a flight reservation Web service offering different operations for searching separately a flight to a destination and its return. We can derive from the first conversation C_1 that the operation pair *(Leave,Return)* is conflicting, i.e., *Return* must be called after *Leave*. The conflict is removed if we have in addition an equivalence relationship $C_1 \sim C_2$. If the WS-RESC document associated to a Web service defines only the C_1 conversation, then the concurrency control manager will delay any attempt to call the operation *Return* until that the operation *Leave* terminates. If C_2 is defined and the equivalence relationship, then invocations can be called in any order. However, calls are executed in mutual exclusion, meaning that whatever is the invocation order, the first interaction should terminate before initiating the second one. Calls can be called concurrently, without being mutually exclusive only if the conversation defines the parallel construct as in Figure V.10 for the above conversations.

Tests are performed for measuring the efficiency gain obtained by by increasing concurrency by allowing concurrent calls to a single Web service from partici-

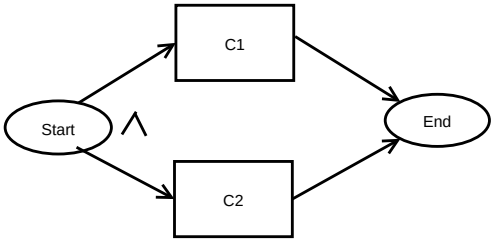


Figure V.10: Concurrent calls

pants of a WSCA operation. A set of automatically generated WSCA executions and Web services have been used.

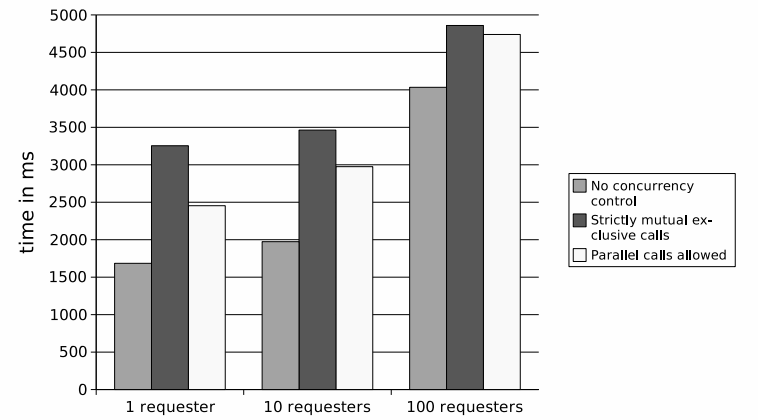


Figure V.11: Measuring parallel access efficiency

Figure V.11 gives the overall performance gain by measuring the total execution time of each WSCA operation. The overall performance gain is however limited by the capabilities of the composed Web services in supporting concurrent accesses. The gain is indeed not perceivable when the composed Web service is over charged due to many concurrent accesses from multiple clients. Further analysis on real Web services is however required to test whether if allowing concurrency induces inconsistencies and hence faults at the client side, i.e., in the composite Web service. This should be considered as a trade-off for a WSCA designer who wants better performances, but should be aware that additional exception handlers can be needed for potential errors that can occur.

V.4.3 Dependability assessment

We assess the dependability of WSCA based composite Web services using measurement techniques. We compare the implementation of the WSCA composite Web service, which uses coordinated exception handling as a means for realizing compensation actions, with an implementation of the WS-BusinessActivity protocol [IBM, Microsoft, BEA, 2004], which is an existent Web service transaction protocol (see Chapter II). Furthermore, a composite Web service written in Java, with no support for reliability is also tested for reference. We use the travel agency composite Web service, but Web services are changed to support the WS-BusinessActivity transaction protocol. The service is implemented in Java, using the Arjuna Transaction Service Suite [10] deployed on top of a JBoss J2EE application server [11].

Several tests have been performed using fault injection techniques to assess the dependability of both systems:

(1) Composed Web services that are accessed have been programmed so that at random intervals WSDL fault messages are sent instead of normal output messages.

(2) Output messages emitted from composed Web services have been altered so that the WSCA operation participant receives unexpected messages.

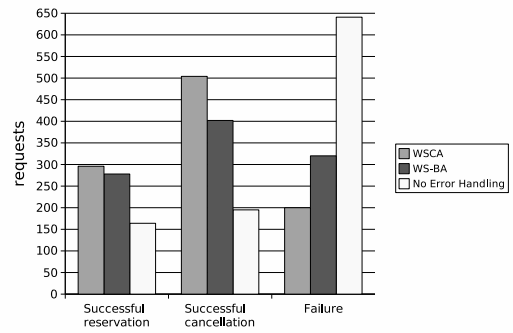
(3) Random delays are introduced in the emission and reception of SOAP messages.

(4) Composed Web services have been made unavailable either by un-deploying the service or by shutting down the application server.

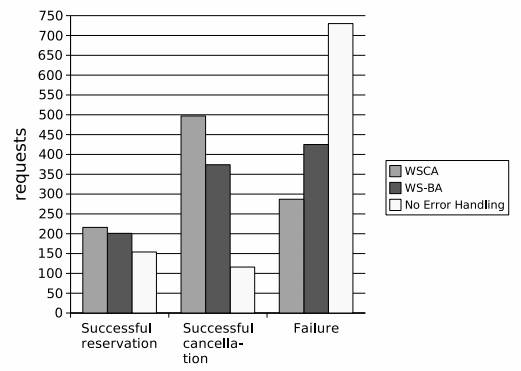
We measured dependability of systems by analyzing the state of the system after 1000 invocations of a composite Web service operation for each test. The number of successful trip reservations, the number of successful cancellation and the number of failures are recorded. The failure is the worst case because the Web services are left in an unknown state (e.g., partially reserved trips). Results are given in the following charts.

In all tests except the third, the WSCA composite Web service results are better than the WS-BusinessActivity implementation. This is mainly due to the fact

[10] http://www.arjuna.com/products/arjunats/index.html
[11] http://www.jboss.com/

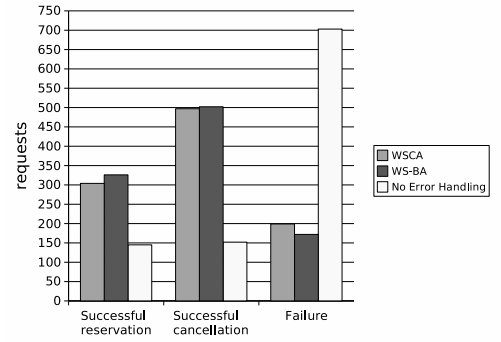


(a) Test 1. Web services send fault messages

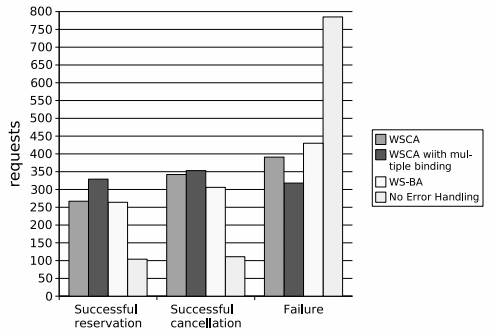


(b) Test 2. Web services send unexpected messages

Figure V.12: Dependability assessments



(a) Test 3. Messages are delayed



(b) Test 4. Web services disappear

Figure V.13: Dependability assessments (2)

that the latter one imposes a tightly coupled interaction with composed Web services, which must follow exactly the transaction protocol. On the other hand, the WSCA composite Web service can deal with unexpected messages by calling an appropriate exception handler. Thanks to the nested structure, exceptions are propagated and never lost. However, it is worth noting that the transaction protocol should be used as a complementary mean for achieving dependability. A composite Web service developed using an orchestration language such as BPEL with error handling can implement the transactional protocol and define other recovery mechanisms for not handled errors. However, mixing recovery protocols in one implementation would add complexity in design. Such a system is then more prone to protocol incompatibilities. The WSCA approach provides an integrated framework to define different recovery strategies, into a well defined structure, easing development of dependable services. Multiple binding is one

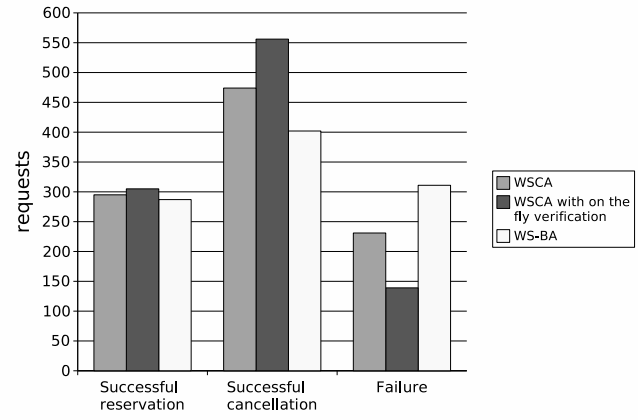


Figure V.14: Dependability assessment with on the fly verification

of the features that can be easily added to the WSCAL specification without increasing complexity, and which increase the overall dependability significantly (Test 4). Furthermore, using the on the fly verification improves dependability by not allowing non-cancellable bookings to be performed (Test 5, realized with randomly selected faults).

V.5 Concluding remarks

In this chapter we have shown that using our proposed languages WSCAL and WS-RESC we can build composite Web services that meet the requirements that we have set. For this purpose, prototype tools have been implemented and tested.

The service discovery protocol that is based on the matching of abstract Web service definitions and on the matching of conversations has been tested. The overall result is that it works relatively well for searching Web services during execution and using dynamic binding of composed Web services. Furthermore, the on the fly verification of invocation correctness increase reliability by preventing faults and the short verification time enables to include the verification on most composite applications. By comparison with existent composition languages, we show that WSCA composite Web services execute more efficiently or with no additional significant overhead. Moreover, increasing performance is possible exploiting the concurrency support of WSCA by relaxing controlled accesses to composed Web services. Compared to existent solutions addressing dependability for Web services, we show that our method is more efficient with a higher

rate for tolerating faults, without losing performance.

VI Conclusion

VI.1 Contribution

Web services are expected to become a major class of systems of systems in the near future. The main objective of this thesis was to enable the integration of autonomous existent Web services into a dependable composite Web service. Our approach primarily lies into: (i) extending the interface of individual Web services in order to reason about their recovery and concurrency capabilities, and (ii) proposing a composition language for specifying a composition process in terms of dependable actions based on forward error recovery.

The specification of recovery support of individual Web services is based on a conversation language (WS-RESC) that allows the specification of both the standard and exceptional behavior of autonomous, composable Web services, further assisting the development of dependable composite services. The main purpose of a conversation language is to define the allowed sequence of interactions that a Web service supports, by specifying dependencies between operations. WS-RESC further includes constructs for specifying concurrency since it is an inherent feature of distributed systems, and for specifying exceptional behaviours, timing constraints and recovery properties of conversation since these are key behavioral properties in the context of dependability. The language in particular enables the definition of equivalence relationships over conversations with respect to their recovery behavior, which is exploited for the design of fault-tolerant composite actions.

The proposed solution for the composition of Web services is based on forward error recovery, oriented towards providing dependability of composite Web services. While exploiting their possible support for fault tolerance (e.g., transactional support at the level of each service), the proposed solution has no impact on the autonomy of the individual Web services. Our solution lies in system structuring in terms of co-operative atomic actions that have a well-defined be-

haviour, both in the absence and in the presence of service failures. More specifically, we defined the notion of Web Service Composition Action (WSCA), based on the Coordinated Atomic Action concept, which allows structuring composite Web services in terms of dependable actions. Fault tolerance is then obtained as an emergent property of the aggregation of several potentially non-dependable services.

The specification of a WSCA is composed of two parts. First, the required interfaces of composed Web services are declared abstractly. Required operations are given using WSDL and required conversations are specified in WS-RESC. These required interfaces are used for service discovery with a dynamic binding mechanism that enables, in particular, to bind Web service instances dynamically at run-time. Furthermore, conversations of composed Web services are used to verify correctness of interactions, thus prevening potential faults. The second part of the WSCA specification comprises the behaviour of each operation offered by the composite Web service. A WSCA operation is defined as a process that comprises several participants, which execute concurrently. Each participant interact with one or more composed Web services. Participants can further join together to form nested WSCA, executing in isolation with each other, with a controlled accesses to composed Web services. Exceptions raised during the execution of a WSCA operation are co-operatively handled by all the participants of the WSCA operation or of a nested WSCA, which terminate with several outcomes. If the coordinated exception handling succeeds, then the WSCA operation or the nested WSCA terminates normally, otherwise, the exception is signalled to a higher-level WSCA or to the service requester as a fault message. We further introduced a framework enabling the development of composite Web services based on WSCAs. The XML-based language for the specification of WSCAs is compiled to Java code, and the WSCA composite Web service is deployed on top of a Web service application server. Intermediary components used by the composite Web service for service discovery and an on the fly verification of the compatiblity of interactions with conversations of Web services are implemented and tested.

As discussed in Section II, there is extensive research work that is ongoing towards supporting the development of fault tolerant composite Web services, relying on the transactional supports for composite Web services. Our contribution primarily comes from relying on forward error recovery instead of backward error recovery for specifying the behavior of composite Web services in the presence of failures and integrating conversation languages in the composition process. Forward error recovery is further specified in terms of co-operative actions. Our analysis shows that this approach is more effective in dealing with faults at the level of composite Web services.

VI.2 Perspectives

The composite Web service development using the WSCAL and WS-RESC languages gives a central role to the developer, which states the requirements of the composite application and specifies the composition process. The requirements are essentially used in the processes of service discovery, verification and concurrency control. Automated analyses are done, for example when constructing the conflict table for customizing the concurrency control. However, the integration of composed Web services can be automated more, for reducing the development time and effort and also for reducing the verification overhead. In particular, in a first stage, the exceptional behaviour expressed in the WS-RESC language can be used to generate correct code skeletons for exception handling. Another improvement for building correct compositions would be the static verification (contrary to the on the fly verification that is implemented) based on a formal model of the composite Web service. Given the formal model of conversations of individual Web services and the one of the composition process, one can formally verify that the implementation of service calls at the service requester-side matches the provided conversation at the service provider-side, in a way similar to architectural connector matching [Allen and Garlan, 1997].

We implemented the local runtime system and associated tools and done expertimentation with use cases. Results shows that the service compositions execute correctly, increasing service dependability. However, as far as dependability is our main concern, a formally specified and implemented runtime system should be considered. In [Tartanoglu et al., 2003b], we have presented an approach for specifying fault tolerance mechanisms using the B formal method. We have considered the use of Coordinated Atomic Actions that have been proved useful for building dependable systems. We have defined a generic formal specification using the B method, defining systems composed of several Coordinated Atomic Actions that make concurrent accesses to external objects. B was chosen because of its powerful theorem proving ability and because of availability of a number of mature tools. We have shown how to specify the following dependability mechanisms of CA Actions: (i) constraints related to the atomic accesses to external transactional objects, (ii) encapsulation of computations inside atomic action units ensured through action nesting and (iii), properties related to the behaviour of the system in case of exception occurrences. This initial specification can be used to define Web Service Composition Actions formally. The main difference would be relaxing atomic accesses to external objects while still preserving isolation, that should be managed locally. In order to have an implementation of the Web Service Composition Action's run-time support, the abstract machines should be refined. At the end of the refinement process, we

will have an executable code that correspond to the implementation of the operations defining the B machines, offered as a programming library. Note that when implementing the WSCA runtime, some existing libraries such as SOAP engines and UDDI API implementations are used. For all these libraries, what is usually known is the interfaces of the offered methods. In order to be able to prove the correctness of the implementation it would be necessary: (i) to have in addition the formal specification of the behaviour of these methods and (ii), to prove that the refinements of the machines that use these methods are correct (in the B sense). During the refinement, the non-determinism will be reduced. The preconditions have to be relaxed in order to take into account all the possible cases. The formal specification together with the refinement process give an executable code that is correct with respect to the specification.

Bibliographie

A. Arkin. Business Process Modeling Language, 2002. http://www.bpmi.org.

R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, 1997.

ANSI. American national standard for information systems. database language sql, November 1992. ANSI X3.135-1992.

BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services, Version 1.1, February 2005. http://www.ibm.com/developerworks/library/specification/ws-bpel/.

BEA Systems, IBM, Microsoft, and TIBCO Software. Web services reliable messaging protocol (ws-reliablemessaging), March 2004. http://www-106.ibm.com/developerworks/library/ws-rm/.

B. Benatallah, F. Casati, and F. Toumani. Web service conversation modeling. *IEEE Internet Computing*, pages 46–54, January-February 2004.

B. Benatallah, M. Dumas, and Q. Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distrib. Parallel Databases*, 17(1) :5–37, 2005. ISSN 0926-8782.

M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. In L. Aceto and B. Victor, editors, *Electronic Notes in Theoretical Computer Science, EXPRRESS'00, 7th International Workshop on Expressiveness in Concurrency*, volume 39. Elsevier, 2003.

A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science*, Mumbai (Bombay), India, December 2003.

Y-D. Bromberg and V. Issarny. Indiss : Interoperable discovery system for networkded services. In *Proceedings of Middleware'2005*, November 2005.

T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification : A New Approach to Design and Analysis of E-Service Composition. In *Proceedings of the 12th International World Wide Web Conference*, May 2003.

R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Operating Systems, Proceedings of an International Symposium*, pages 89–102, London, UK, 1974. Springer-Verlag. ISBN 3-540-06849-X.

R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *Transactions on Software Engineering*, SE-12(8) :811–826, 1986.

F. Cristian. *Dependability of Resilient Computers*, chapter Exception Handling, pages 68–97. Blackwell Scientific Publications, 1989.

A. Elfatatry and P. Layzell. Negotiating in service-oriented environments. *CACM*, 47(8) :103–108, August 2004.

A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

W. Emmerich. *Engineering Distributed Objects*. J. Wiley & Sons, 2000.

F. Leymann, 2001. Web Services Flow Language (WSFL 1.0), may 2001. IBM, `http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

R. Farahbod, U. Glasser, and M. Vajihollahi. *ASM 2004, W. Zimmermann and B. Thalheim editors, LNCS 3052*, chapter Specification and Validation of the Business Process Execution Language for Web Services, pages 78–94. Springer-Verlag, 2004.

A. Ferrara. Web services : a process algebra approach. In *ICSOC '04 : Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-871-7.

H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *ICWS '04 : Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 738, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2167-3.

S. Frolund and K. Govindarajan. cl : A language for formally defining web services interactions. Technical report, HP Laboratories Palo Alto, October 2003.

X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04 : Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-844-X.

H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8 :186–213, 1983.

H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data*, 1987.

M.C. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. DSoS Conceptual Model. Technical report, IST Project Dependable Systems of Systems, IST-1999-11585, 2003.

J B. Goodenough. Exception handling : issues and a proposed notation. *Commun. ACM*, 18(12) :683–696, 1975. ISSN 0001-0782.

J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, 1993.

R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *CRPITS'17 : Proceedings of the Fourteenth Australasian database conference on Database technologies 2003*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. ISBN 0-909-92595-X.

J. E. Hanson, P. Nandi, and D. Levine. Conversation-enabled web services for agents and e-business. In *Proceedings of the International Conference on Internet Computing (IC-02)*, pages 791–796. CSREA Press, 2002.

IBM. Httpr specification version 1.1, April 2002. `http://www.ibm.com/developerworks/library/ws-httprspec/`.

IBM, Microsoft, BEA. Web services businessactivity framework, 2004. `http://www-106.ibm.com/developerworks/webservices/library/ws-busact/`.

IETF. Rfc 821, simple mail transfer protocol, 1982. `http://www.ietf.org/rfc/rfc0821.txt`.

IETF. Rfc 2396, unifotm resource identifiers (uri) : Generic syntax, 1998. `http://www.ietf.org/rfc/rfc2396.txt`.

IETF, 1999. Rfc 2616, hypertext transfer protocol – http/1.1, ietf, 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

V. Issarny. An exception handling mechanism for parallel object-oriented programming : Towards reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6) :29–39, 1993.

R. Jimenez-Peris, M. Patino-Martinez, S. Woodman, S. Shrivastava, D. Palmer, S. Wheater, B. Kemme, and G. Alonso. Service specification language,. Technical report, Deliverable of ADAPT IST project IST-2001-37126, 2003.

H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

P. A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer - Verlag, 2nd edition, 1990.

A. Martens. *Lecture Notes in Computer Science*, volume 3442, chapter Analyzing Web Service Based Business Processes, pages 19–33. Springer-Verlag, 2005.

G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Tech. J.*, 34 :1045–1079, September 1955.

B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal*, 12(4) :333–351, 2003. ISSN 1066-8888.

L.G. Meredith and S. Bjorg. Contracts and types. *Communications of the ACM*, 46(10) :41–47, October 2003.

Microsoft, BEA and IBM. Web Services Business Activity Framework (WS-BusinessActivity), November 2004a. `http://www.ibm.com/developerworks/library/ws-transpec/`.

Microsoft, BEA and IBM. Web Services Coordination (WS-Coordination), November 2004b. `http://www.ibm.com/developerworks/library/ws-coor/`.

T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes : Reliable composition of autonomous Web services. In *DSN 2002, Workshop on Dependable Middleware-based Systems (WDMS 2002)*, 2002.

R. Miller and A. Tripathi. *P. Ezhilchelvan, A. Romanovsky, editors, Concurrency in Dependable Computing*, chapter Exception Handling in Timed Asynchronous Systems, pages 209–227. Kluwer, 2002.

R. Milner. *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, 1999.

S. Narayanan and S. McIlraith. Simulation, verification and automated composition of Web services. In *Proceedings of the WWW'02 Conference*, 2002.

OASIS. Business Transaction Protocol (BTP), Version 1.1, 2004a. `http://www.oasis-open.org/committees/business-transactions/`.

OASIS. OASIS Wes Services Reliability (WS-Reliability), 2004b. OASIS Working Draft, `http://www.oasis-open.org`.

OASIS. UDDI, Version 3, API Specification, 2004c. http ://www.uddi.org.

OMG. The common object request broker 3.0 - omg idl syntax and semantics chapter. Technical Report 02-06-39, OMG Document, 2002. http ://http.omg.org.

M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Commun. ACM*, 46(10), 2003.

P. F. Pires, M. R. F. Benevides, and M. Mattoso. Mediating heterogeneous web services. In *SAINT '03 : Proceedings of the 2003 Symposium on Applications and the Internet*, page 344, Washington, DC, USA, 2003a. IEEE Computer Society. ISBN 0-7695-1872-9.

P.F. Pires, M. Benevides, and M. Mattoso. *Web, Web-Services, and Database Systems 2002*, chapter Building Reliable Web Services Compositions, pages 59–72. Springer LNCS 2593, 2003b.

C. Pu, G. Kaiser, and N. Hutchinson. Split-transaction for open-ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 26–37, 1988.

B. Randell. Recursive structured distributed computing systems. In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, 1983.

W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I : Basic Models*, volume 1491. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1998.

D. Sangiorgi and D. Walker. *The pi-calculus : A Theory of Mobile Processes*. Cambridge Universtity Press, 2001.

K. Schmidt and C. Stahl. A petri net semantic for bpel. In *Proceedings of the 11th Workshop AWPN*, October 2004.

S. Tai, T. Mikalsen, E. Wohlstadter, N. Desai, and I. Rouvellou. Transaction policies for service-oriented computing. *Data Knowl. Eng.*, 51(1) :59–79, 2004. ISSN 0169-023X.

F. Tartanoglu and V. Issarny. Specifying web services recovery support with conversations. In *Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS'2005)*, January 2005.

F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the Web Service Architecture. In *Architecting Dependable Systems, LNCS 2677*, pages 89–108. Springer-Verlag, 2003a.

F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Formalizing Dependability Mechanisms in B : From Specification to Development Support. In *Proceedings of the ICSE Workshop on Architecting Dependable Systems*, Portland, USA, May 2003b.

F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated Forward Error Recovery for Composite Web Services. In *Proceedings of the* 22^{nd} *IEEE Symposium on Reliable Distributed Systems*, pages 167–176, Florence, Italy, October 2003c.

F. Tartanoglu, N. Levy, V. Issarny, and A. Romanovsky. Using the B Method for the Formalization of Coordinated Atomic Actions, October 2004. CS-TR 865, Department of Computing Science, University of Newcastle upon Tyne.

S. Thatte. Xlang : Web services for business process design, 2001. Microsoft Corporation, `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/`.

R. Tolksdorf. A dependency markup language for web services. In *Web Databases and Web Services 2002*, pages 129–140, 2003. LNCS 2593.

W3C. XML Path Language (XPath), Version 1.0, 1999. W3C Recommendation, `http://www.w3.org/TR/xpath`.

W3C. Web Service Choreography Interface (WSCI) 1.0, W3C Note, 2002a. `http://www.w3.org/TR/wsci/`.

W3C. Web services conversation language (WSCL), version 1.0, 2002b. W3C Note, `http://www.w3.org/TR/wscl10/`.

W3C. Owl-s : Semantic markup for web service, 2003a. `http://www.daml.org/services/owl-s/`.

W3C. Soap version 1.2, 2003b. W3C Recommendation, `http://www.w3.org/2000/xp/Group/`.

W3C. Extensible markup language (xml) 1.1, 2004a. W3C Recommendation, `http://www.w3.org/TR/xml11`.

W3C. Web services choreography description language version 1.0, 2004b. W3C Working Draft, `http://www.w3.org/TR/ws-cdl-10/`.

W3C. Xml schema, 2004c. W3C Recommandation, `http://www.w3.org/XML/Schema`.

W3C. Web services description language (WSDL), version 2.0, 2005. W3C Working Draft, `http://www.w3.org/TR/wsdl20/`.

J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth IEEE International Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.

J. Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(10) :1019–1032, 2000.

X. Yi and K.J. Kochut. Process composition of Web services with complex conversation protocols : a colored Petri nets based approach. In *Proceedings of Advanced Simulation Technology Conference DASD2004*, Arlington, Virginia, USA, April 2004.

Annexes

A WS-RESC XML Schema definition

This appendix presents the definition of the WS-RESC language introduced in Chapter III using the XML Schema language [W3C, 2004c]. XML Schema is a definition language used for describing and constraining the content of XML documents. An equivalent graphical representation is also given for illustration.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="wsresc">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:choice>
          <xs:element name="state">
            <xs:complexType>
              <xs:attribute name="name" type="xs:NCName"
                 use="required"/>
              <xs:attribute name="operation" type="xs:QName"
                 use="required"/>
              <xs:attribute name="correlate" type="xs:QName"
                 use="optional"/>
            </xs:complexType>
          </xs:element>
          <xs:element name="activity">
            <xs:complexType>
              <xs:sequence maxOccurs="unbounded">
                <xs:choice>
                  <xs:element name="transition">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="source"
                           maxOccurs="unbounded">
                          <xs:complexType>
                            <xs:attribute name="state" type="xs:QName"
```

```
                use="required"/>
              <xs:attribute name="condition"
                 type="xs:anySimpleType" use="optional"/>
            </xs:complexType>
          </xs:element>
          <xs:choice>
            <xs:element name="concurrent">
              <xs:complexType>
                <xs:sequence>
                  <xs:element ref="destination"
                     maxOccurs="unbounded"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element ref="destination"
               maxOccurs="unbounded"/>
          </xs:choice>
          <xs:element name="timeout">
            <xs:complexType>
              <xs:attribute name="timer"
                 type="xs:duration" use="required"/>
              <xs:attribute name="onInput"
                 type="xs:boolean" use="optional"
                 default="false"/>
              <xs:attribute name="state" type="xs:QName"
                 use="required"/>
              <xs:attribute name="exception" type="xs:QName"
                 use="optional"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName"
           use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="exception">
      <xs:complexType>
        <xs:attribute name="name" type="xs:NCName"
           use="required"/>
        <xs:attribute name="condition"
           type="xs:anySimpleType" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="handler">
      <xs:complexType>
        <xs:attribute name="exception" type="xs:QName"
           use="required"/>
        <xs:attribute name="activity"
           type="xs:QName" use="required"/>
      </xs:complexType>
```

```
                </xs:element>
                <xs:element name="property">
                  <xs:complexType>
                    <xs:attribute name="value" type="xs:QName"
                       use="required"/>
                  </xs:complexType>
                </xs:element>
              </xs:choice>
            </xs:sequence>
            <xs:attribute name="name" type="xs:NCName"
              use="required"/>
            <xs:attribute name="ref" type="xs:QName"
              use="optional"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="equivalence">
          <xs:complexType>
            <xs:sequence minOccurs="2" maxOccurs="unbounded">
              <xs:element name="equiv">
                <xs:complexType>
                  <xs:attribute name="activity" type="xs:QName"
                     use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="destination">
   <xs:complexType>
     <xs:attribute name="state" type="xs:QName" use="required"/>
     <xs:attribute name="minOccurs"  type="xs:nonNegativeInteger"
        use="optional" default="1"/>
     <xs:attribute name="maxOccurs" type="xs:nonNegativeInteger"
        use="optional" default="1"/>
     <xs:attribute name="condition" type="xs:anySimpleType"
        use="optional"/>
   </xs:complexType>
 </xs:element>
</xs:schema>
```

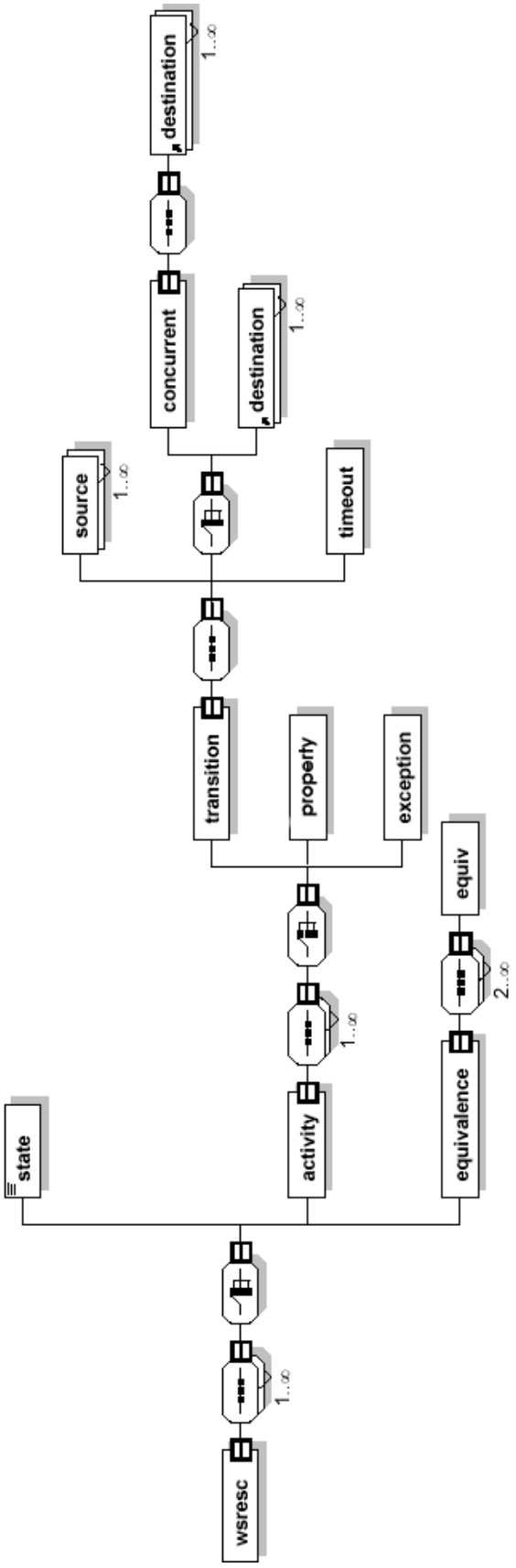


Figure A.1 – WS-RESC

B WSCAL XML Schema definition

This appendix presents the definition of the WSCAL language introduced in Chapter IV using the XML Schema language [W3C, 2004c] together with an equivalent graphical representation for illustration.

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="WSCAL">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="state" type="stateType" minOccurs="0"/>
        <xs:element name="services">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="service" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="staticService" minOccurs="0"
                       maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="hrefSchema"
                           type="xs:anyURI" use="required"/>
                        <xs:attribute name="onCall" type="xs:boolean"
                           use="optional" default="false"/>
                        <xs:attribute name="multiple" type="xs:boolean"
                           use="optional" default="false"/>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="dynamicService" minOccurs="0"/>
                  </xs:sequence>
                  <xs:attribute name="hrefSchema"
                    type="xs:anyURI" use="required"/>
                  <xs:attribute name="conversation"
                    type="xs:anyURI" use="optional"/>
```

```
                <xs:attribute name="strict"
                  type="xs:boolean" use="optional"
                  default="false"/>
                <xs:attribute name="isolation"
                  type="xs:QName" use="optional"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="WSCA" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="input" type="messageType"
               minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="output" type="messageType"
               minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="fault" type="messageType"
               minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="state" type="stateType"
               minOccurs="0"/>
            <xs:element name="before" type="behaviourType"/>
            <xs:element name="after" type="behaviourType"/>
            <xs:element name="abort" type="behaviourType"/>
            <xs:element name="nested" type="nestedType"/>
            <xs:element name="participant" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="state" type="stateType"/>
                  <xs:element name="behaviour" type="behaviourType"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="operation" type="xs:QName"
             use="required"/>
          <xs:attribute name="exceptionTree" type="xs:anyURI"
             use="optional"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="scope" type="scopeType" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="scopeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="request"/>
    <xs:enumeration value="application"/>
    <xs:enumeration value="session"/>
```

```
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="messageType">
  <xs:attribute name="message" type="xs:NCName" use="required"/>
</xs:complexType>
<xs:complexType name="stateType">
  <xs:sequence>
    <xs:element name="types">
      <xs:complexType>
        <xs:sequence>
          <xs:any namespace="##other" minOccurs="0"
             maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="var" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="name" type="xs:NCName" use="required"/>
        <xs:attribute name="type" type="xs:QName" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="behaviourType">
  <xs:sequence>
    <xs:element name="try" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="Statements" maxOccurs="unbounded"/>
          <xs:element name="coordinatedHandler"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:group name="Statements">
  <xs:choice>
    <xs:element name="sequence">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="Statements" minOccurs="0"
             maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="all">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="Statements" minOccurs="0"
             maxOccurs="unbounded"/>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="switch">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="case" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="Statements" minOccurs="0"
               maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="condition" type="xs:boolean"
             use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="default">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="Statements" minOccurs="0"
               maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="while">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="Statements" minOccurs="0"
         maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="condition" type="xs:boolean"
       use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="try">
  <xs:complexType>
    <xs:sequence>
      <xs:group ref="Statements" minOccurs="0"
         maxOccurs="unbounded"/>
      <xs:element name="catch" minOccurs="0"
         maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:group ref="Statements" minOccurs="0"
               maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="exception" type="xs:QName"
```

```
                use="optional"/>
              <xs:attribute name="exceptionData" type="xs:NCName"
                use="optional"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="optional"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="call" type="callType"/>
    <xs:element name="assign" type="assignType"/>
    <xs:element name="empty" type="emptyType"/>
    <xs:element name="wait" type="waitType"/>
    <xs:element name="join" type="joinType"/>
    <xs:element name="syncPoint" type="syncPointType"/>
    <xs:element name="return" type="returnType"/>
    <xs:element name="throw" type="throwType"/>
  </xs:choice>
</xs:group>
<xs:complexType name="callType">
  <xs:sequence>
    <xs:element name="input" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="message" type="xs:NCName"
          use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="output" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="message" type="xs:NCName"
          use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="fault" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="message" type="xs:NCName"
          use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="onAbort" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="Statements" minOccurs="0"
            maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="onTimeout" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
```

```
          <xs:group ref="Statements" minOccurs="0"
             maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
  <xs:attribute name="service" type="xs:QName" use="required"/>
  <xs:attribute name="operation" type="xs:QName" use="required"/>
  <xs:attribute name="abortable" type="xs:boolean" use="optional"
     default="true"/>
  <xs:attribute name="retry" type="xs:boolean" use="optional"
     default="false"/>
  <xs:attribute name="tryAlternate" type="xs:boolean"
     use="optional" default="false"/>
  <xs:attribute name="timeout" type="xs:duration" use="optional"/>
</xs:complexType>
<xs:complexType name="assignType">
  <xs:sequence>
    <xs:element name="to">
      <xs:complexType>
        <xs:attribute name="var" type="xs:QName" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="from">
      <xs:complexType>
        <xs:attribute name="var" type="xs:QName" use="required"/>
        <xs:attribute name="expr" type="xs:anySimpleType"
           use="optional"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
</xs:complexType>
<xs:complexType name="emptyType">
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
</xs:complexType>
<xs:complexType name="waitType">
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
  <xs:attribute name="type" type="waitTypeType" use="optional"
     default="duration"/>
  <xs:attribute name="timer" type="xs:duration" use="optional"/>
</xs:complexType>
<xs:simpleType name="waitTypeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="duration"/>
    <xs:enumeration value="absoluteTime"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="joinType">
```

```
  <xs:sequence>
    <xs:element name="condition" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="action" type="xs:QName"
           use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="timeout" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="Statements" minOccurs="0"
             maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="type" type="waitTypeType"
           use="optional" default="duration"/>
        <xs:attribute name="timer" type="xs:duration"
           use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="onTimeout" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
</xs:complexType>
<xs:complexType name="syncPointType">
  <xs:sequence>
    <xs:element name="timeout" minOccurs="0">
      <xs:complexType>
        <xs:attribute name="type" type="waitTypeType"
           use="optional" default="duration"/>
        <xs:attribute name="timer" type="xs:duration"
           use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="onTimeout" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:group ref="Statements" minOccurs="0"
             maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
  <xs:attribute name="count" type="xs:nonNegativeInteger"
     use="required"/>
</xs:complexType>
<xs:complexType name="returnType">
  <xs:attribute name="name" type="xs:NCName" use="optional"/>
  <xs:attribute name="value" type="xs:QName" use="optional"/>
  <xs:attribute name="expr" type="xs:anySimpleType"
```

```
      use="optional"/>
  </xs:complexType>
  <xs:complexType name="throwType">
    <xs:attribute name="name" type="xs:NCName" use="optional"/>
    <xs:attribute name="exception" type="xs:QName" use="required"/>
    <xs:attribute name="exceptionData" type="xs:NCName"
       use="optional"/>
  </xs:complexType>
  <xs:complexType name="nestedType">
    <xs:sequence>
      <xs:element name="before" type="behaviourType"/>
      <xs:element name="after" type="behaviourType"/>
      <xs:element name="abort" type="behaviourType"/>
      <xs:element name="participant" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="pname" type="xs:QName"
             use="required"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="nested" type="nestedType"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
  </xs:complexType>
</xs:schema>
```

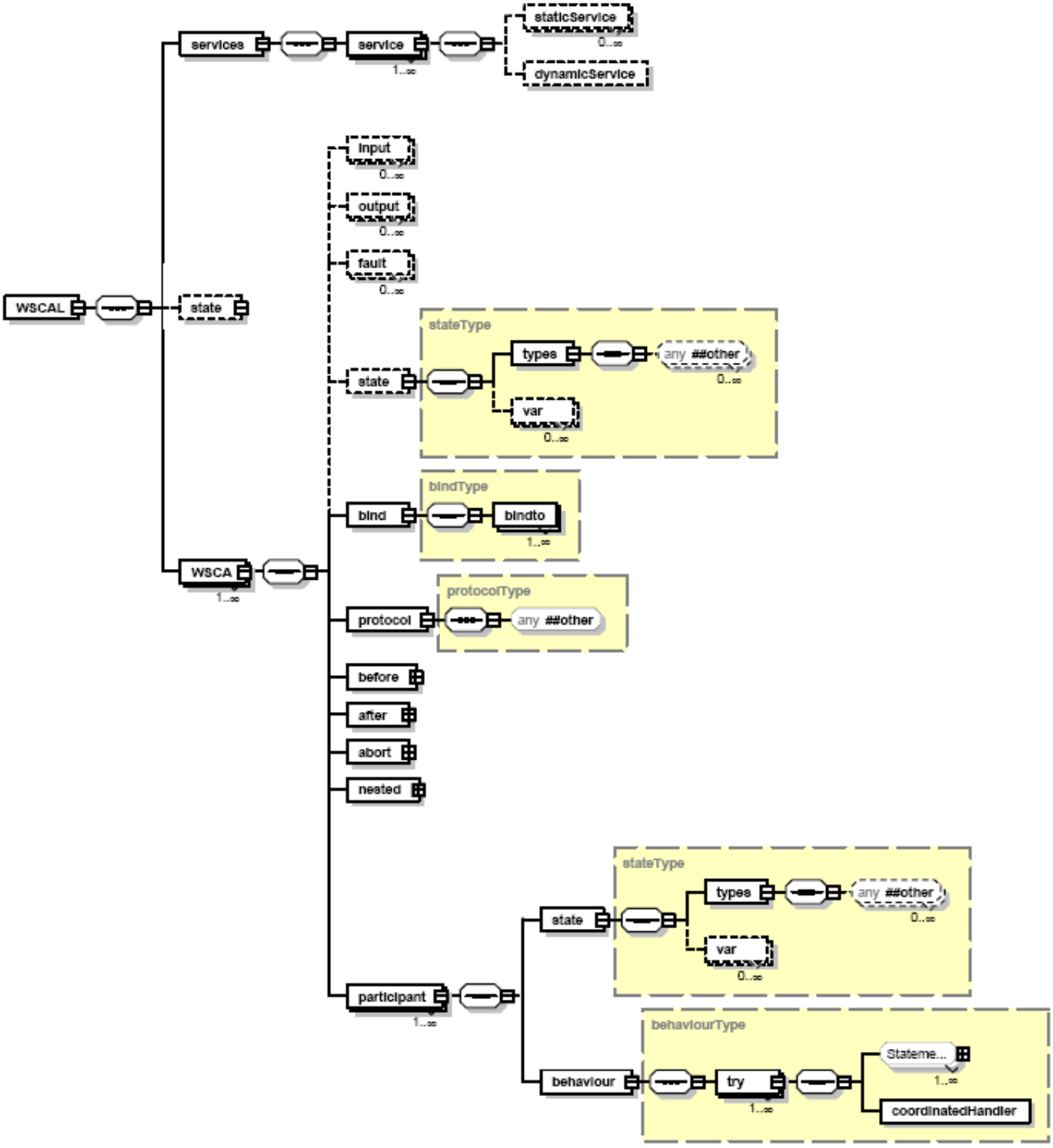


Figure B.1 – WSCAL

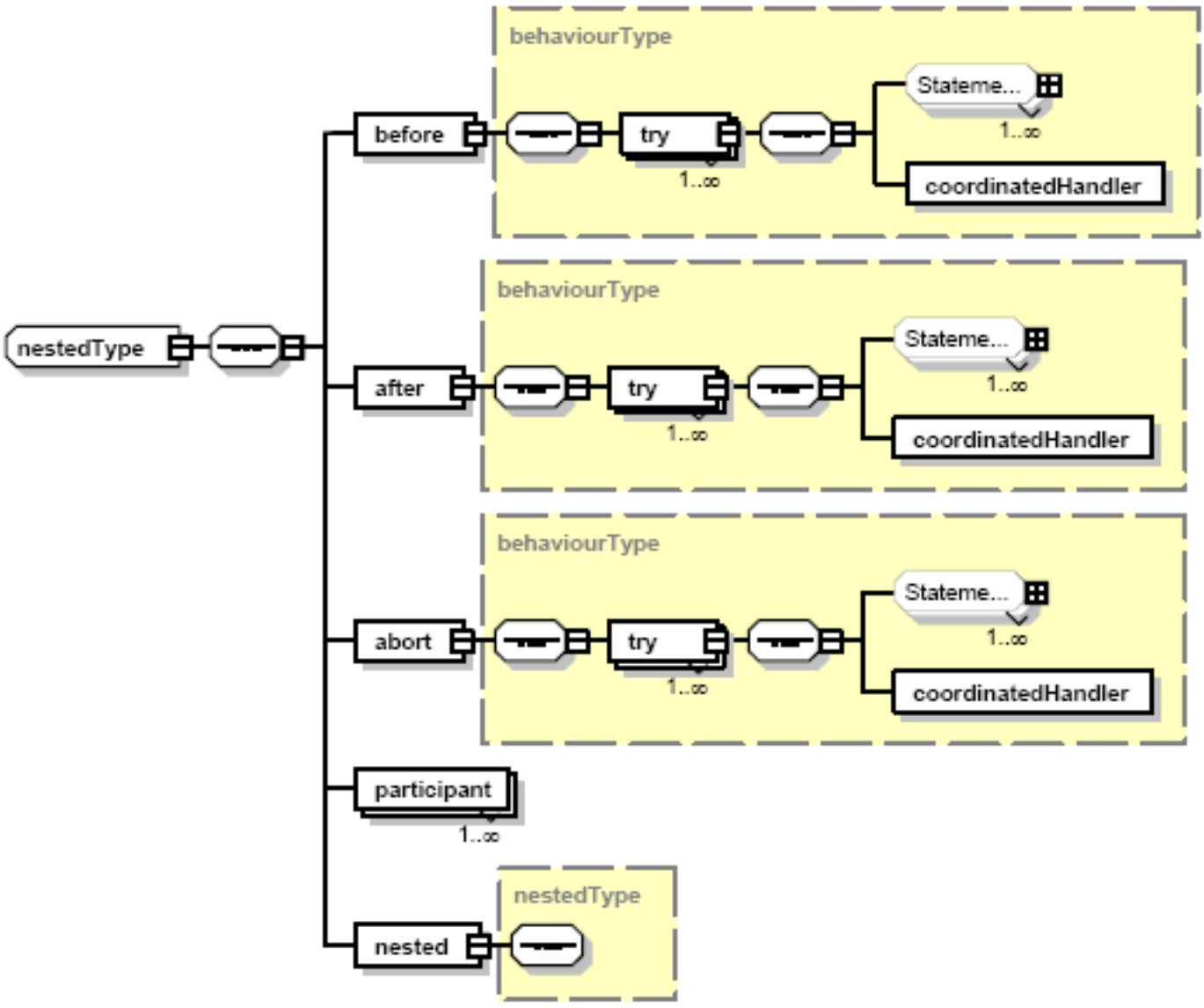


Figure B.2 – Nested WSCA

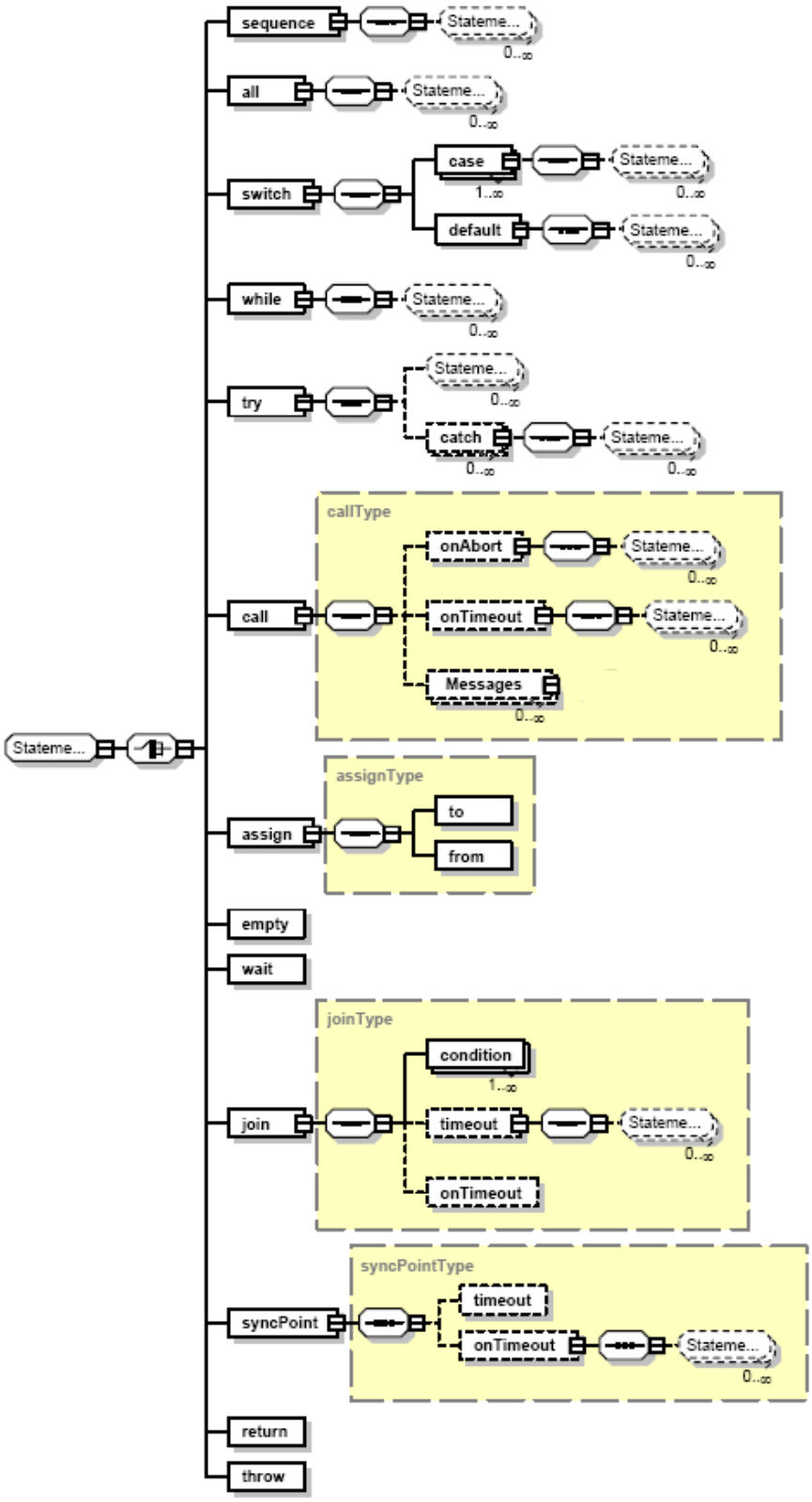


Figure B.3 – WSCAL Statements

C Travel agency WSCAL listing

In this appendix we give the full listing of the travel agent composite Web service in WSCAL, used in the experiments in Chapter V.

```
<WSCAL
  targetNamespace="http://travelagency.com"
  xmlns:ta="http://travelagency.com"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  xmlns:wsdl_f="http://travelagency.com/flight.wsdl"
  xmlns:wsdl_h="http://travelagency.com/hotel.wsdl"

  name="TravelAgent" scope="request">

  <services>
    <service name="HotelService"
        hrefSchema="http://travelagency.com/hotel.wsdl"
        conversation="http://travelagency.com/hotel.resc"
        strict=false
        isolation=none >
      <staticService hrefSchema="http://hotels.com/webservices/service.wsdl"/>
      <staticService hrefSchema="http://accomodation.com/ws.wsdl"/>
    </service>

    <service name="FlightService"
        hrefSchema="http://travelagency.com/flight.wsdl"
        conversation="http://travelagency.com/flight.resc"
        strict=false
        isolation=none >
      <dynamicService onCall=false multiple=true />
    </service>

  <WSCA operation="bookTrip" exceptionRules="http://travelagency.com/rules.xml">

    <input message="wsdl:tripRequest" name="tripRequest" />
    <output message="wsdl:tripResponse" name="tripResponse" />
```

```
<fault message="wsdl:noTripAvailable" name="noTrip" />
<fault message=wsdl:Failure name="failed"/>
<state>
  <type>
    <xsd:complexType name="bookTripResponseType">
      <xsd:sequence>
        <xsd:element name="hotelDetail" type="xsd:string"/>
        <xsd:element name="flightDetail" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </type>

  <var name="bookTripResponse" type="ta:bookTripResponseType/>
</state>

<after>
  <assign>
    <to var="ta:bookTripResponse/hotelDetail"/>
    <from var="ta:hotelConfirmationResponse"/>
  </assign>
  <assign>
    <to var="ta:bookTripResponse/flightDetail"/>
    <from var="ta:flightConfirmationResponse"/>
  </assign>
  <assign>
    <to var="ta:tripRequest"/>
    <from var="ta:bookTripResponse"/>
  </assign>
</after>

<abort>
  <switch>
    <case condition="ta:noTripAvailable">
      <assign>
        <to var="ta:noTrip"/>
        <from var="ta:noTripAvailable/@exceptionData"/>
      </assign>
    </case>
    <case condition="ta:failure">
      <assign>
        <to var="ta:failed"/>
        <from var="ta:"abort/@exceptionData"/>
      </assign>
    </case>
  </switch>
</abort>

<nestedWSCA name="search"
    exceptionRules="http://travelagency.com/search.xml">
```

```
      <participant name="ta:Hotel" />
      <participant name="ta:Flight" />
    </nested>

    <nestedWSCA name="book"
        exceptionRules="http://travelagency.com/bookHotel.xml">

      <participant name="ta:Hotel" />
      <participant name="ta:Flight" />
    </nested>

    <participant name="Hotel">

      <state>
        <var name="hotelDestination" type="xsd:string"/>
<var name="hotelArrival" type="xsd:date"/>
<var name="hotelDeparture" type="xsd:date"/>
<var name="hotelRooms" type="xsd:string"/>
        <var name="RoomNotAvailable" type="xsd:string"/>
        <var name="hotelConfirmationResponse" type="xsd:string"/>
        <var name="hotelCancelConfirmationResponse" type="xsd:string"/>
      </state>

      <behaviour>
        <try>
          <sequence>
             <assign>
                <to var="ta:hotelDestination"/>
                <from var="ta:tripRequest"
                    expr="//destination">
             </assign>
             <assign>
               <to var="ta:hotelDeparture"/>
               <from var="ta:tripRequest"
                   expr="//departure">
             </assign>
             <assign>
               <to var="ta:hotelArrival"/>
               <from var="ta:tripRequest"
                   expr="//arrival">
             </assign>
             <startNested="ta:search">
               <sequence>
                 <try>
                   <sequence>
                     <call name="invokeHotel" service="ta:HotelService"
                           operation="searchAvailability"
                           retry=1
                           tryAlternate=true >
                       <input message="wsdl:hotelInput">
```

```
                    <ta:hotelDestination/>
                    <ta:hotelDeparture/>
                    <ta:hotelArrival/>
                  </input>
                  <output message="wsdl:hotelOutput">
                    <ta:hotelRooms/>
                  </output>
                  <fault message="wsdl:hotelFault">
                    <ta:RoomNotAvailable/>
                  </fault>
                </call>
                <switch>
                  <case condition="ta:RoomNotAvailable">
                    <throw exception="ta:RoomNotFound" />
                  </case>
                </switch>
                <catch exception="ta:RoomNotFound">
                  <try>
                    <sequence>
                      <call ref="invokeHotel" service="ta:HotelService[2]"/>
                      <switch>
                        <case condition="ta:RoomNotAvailable">
                          <throw exception="ta:HotelSearchFailed" />
                        </case>
                      </switch>
                    </sequence>
                  </try>
                </catch>
              </sequence>
            </try>
            <coordinatedHandler exception="ta:Failed">
              <thow exception="ta:noTripAvailable"
                    exceptionData="noTripAvailable"/>
            </coordinatedHandler>
        </startNested>

        <startNested="ta:bookHotel">
           <try>
             <sequence>
                <call name="bookHotel" service="ta:HotelService"
                      operation="Book"
                      retry=1
                      tryAlternate=false >
                  <input message="wsdl:hotelInput">
                    <ta:hotelDestination/>
                    <ta:hotelDeparture/>
                    <ta:hotelArrival/>
                  </input>
                  <output message="wsdl:hotelConfirmation">
                    <ta:hotelConfirmationResponse/>
```

```
                    </output>
                    <fault message="wsdl:hotelFault">
                      <ta:HotelBookFailed/>
                    </fault>
                  </call>
                  <switch>
                    <case condition="ta:HotelBookFailed">
                      <throw exception="ta:HotelBookFailed" />
                    </case>
                  </switch>
                </sequence>
              </try>
              <coordinatedHandler exception="ta:flightBookFailed">
                 <sequence>
                   <call name="cancelHotel" service="ta:HotelService"
                         operation="Cancel"
                         retry=1
                         tryAlternate=false >
                    <input message="wsdl:hotelCancelInput">
                      <ta:hotelConfirmationResponse/>
                    </input>
                    <output message="wsdl:hotelCancelConfirmation">
                      <ta:hotelCancelConfirmationResponse/>
                    </output>
                    <fault message="wsdl:hotelFault">
                      <ta:HotelCancelFailed/>
                    </fault>
                  </call>
                  <switch>
                    <case condition="ta:HotelCancelFailed">
                      <throw exception="ta:HotelCancelFailed" />
                    </case>
                  </switch>
                </sequence>
              </coordinatedHandler>
            </startNested>
          </sequence>
        </try>
        <coordinatedHandler exception="failure">
           <throw exception="abort" exceptionData="Aborted" />
        </coordinatedHandler>
      </behaviour>
    </participant>

    <participant name="Flight">
      <state>
        <var name="flightDestination" type="xsd:string"/>
        <var name="flightFrom" type="xsd:string"/>
<var name="flightArrival" type="xsd:date"/>
<var name="flightDeparture" type="xsd:date"/>
```

```
<var name="flightSeats" type="xsd:string"/>
        <var name="flightConfirmationResponse" type="xsd:string"/>
        <var name="flightCancelResponse" type="xsd:string"/>
      </state>

      <behaviour>
        <try>
          <sequence>
            <assign>
              <to var="ta:flightDestination"/>
              <from var="ta:tripRequest"
                  expr="//destination">
            </assign>
            <assign>
              <to var="ta:flightFrom"/>
              <from var="ta:tripRequest"
                  expr="//from">
            </assign>
            <assign>
              <to var="ta:flightDeparture"/>
              <from var="ta:tripRequest"
                  expr="//departure">
            </assign>
            <assign>
              <to var="ta:flightArrival"/>
              <from var="ta:tripRequest"
                  expr="//arrival">
            </assign>
            <startNested="ta:search">
               <sequence>
                 <try>
                   <sequence>
                     <call name="invokeFlight" service="ta:FlightService"
                           operation="wsdl_f:searchFlight"
                           retry=1
                           tryAlternate=true >
                       <input message="wsdl_f:flightInput">
                         <ta:flightDestination/>
                         <ta:flightFrom/>
                         <ta:flightDeparture/>
                         <ta:flightArrival/>
                       </input>
                       <output message="wsdl_f:flightOutput">
                         <ta:flightSeats/>
                       </output>
                       <fault message="wsdl_f:flightFault">
                         <ta:FlightNotAvailable/>
                       </fault>
                     </call>
                     <switch>
```

```
                <case condition="ta:FlightNotAvailable">
                  <throw exception="ta:FlightNotFound" />
                </case>
              </switch>
              <catch exception="ta:FlightNotFound">
                <try>
                  <sequence>
                    <call ref="invokeFlight"
                          service="ta:FlightService[2]"/>
                    <switch>
                      <case condition="ta:FlightNotAvailable">
                        <throw exception="ta:FlightSearchFailed" />
                      </case>
                    </switch>
                  </sequence>
                </try>
              </catch>
            </try>
            <coordinatedHandler exception="ta:Failed">
              <thow exception="ta:noTripAvailable"
                    exceptionData="noTripAvailable"/>
            </coordinatedHandler>
        </startNested>

        <startNested="ta:bookFlight">
            <try>
              <sequence>
                 <call name="bookFlight" service="ta:flightService"
                       operation="Book"
                       retry=1
                       tryAlternate=false >
                  <input message="wsdl:FlightInput">
                    <ta:flightDestination/>
                    <ta:flightFrom/>
                    <ta:flightDeparture/>
                    <ta:flightArrival/>
                  </input>
                  <output message="wsdl:flightConfirmation">
                    <ta:flightConfirmationResponse/>
                  </output>
                  <fault message="wsdl:FlightFault">
                    <ta:flightBookFailed/>
                  </fault>
                </call>
                <switch>
                  <case condition="ta:flightBookFailed">
                    <throw exception="ta:flightBookFailed" />
                  </case>
                </switch>
              </sequence>
```

```
                </try>
                <coordinatedHandler exception="ta:hotelBookFailed">
                   <sequence>
                     <call name="cancelFlight" service="ta:FlightService"
                           operation="Cancel"
                           retry=1
                           tryAlternate=false >
                       <input message="wsdl:flightCancelInput">
                         <ta:flightConfirmationResponse/>
                       </input>
                       <output message="wsdl:flightCancelConfirmation">
                         <ta:flightCancelConfirmationResponse/>
                       </output>
                       <fault message="wsdl:flightFault">
                         <ta:FlightCancelFailed/>
                       </fault>
                     </call>
                     <switch>
                       <case condition="ta:FlightCancelFailed">
                         <throw exception="ta:FlightCancelFailed" />
                       </case>
                     </switch>
                   </sequence>
                </coordinatedHandler>
             </startNested>
           </sequence>
         </try>
         <coordinatedHandler exception="failure">
            <throw exception="abort"/>
         </coordinatedHandler>
       </behaviour>
     </participant>
  </WSCA>
</WSCAL>
```